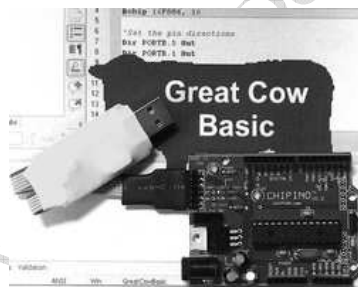


Great Cow Basic CHIPINO



Programming Manual

Published by Electronic Products.

This is an online version for Personal use only.

Please do not copy.

The publisher offers special discounts on bulk orders of this book.

For information contact:

Electronic Products

P.O. Box 251

Milford, MI 48381

www.elproducts.com

chuck@elproducts.com

The Microchip name and logo, MPLAB® and PIC® are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. chipKIT™ is a trademark of Microchip Technology Inc. in the U.S.A. and other countries.

All other trademarks mentioned herein are the property of their respective companies.

Printed in the United States of America

Table of Contents

1.0 Introduction.....	7
1.1 Using Great Cow Basic.....	9
1.2 Using Great Cow Basic.....	13
1.3 Creating your first project.....	15
1.4 Command Line Options.....	20
2.0 Compiler Basics.....	21
2.1 Inputs/Outputs.....	21
2.3 Variables.....	24
2.4 Casting.....	28
2.5 Defines/Constants.....	28
2.6 Numeric Constants.....	30
2.6.1 BCD to Decimal.....	30
2.6.2 Decimal to BCD.....	30
2.7 String Constants.....	31
2.8 String Conversion.....	32
2.8.1 String to Hex.....	32
2.8.2 String to Decimal Value.....	32
2.8.3 Find String within a String.....	32
2.8.4 Convert to Lower Case.....	33
2.8.5 Convert to Upper Case.....	33
2.8.6 Convert Number to String.....	34
2.8.7 Retrieve String Characters (Left).....	34
2.8.8 Retrieve String Characters (Middle).....	35
2.8.9 Retrieve String Characters (Right).....	35
2.9 Mathematical Operations.....	35
2.10 Conditions.....	37
2.11 Combining Multiple Instructions.....	38
2.12 Comments.....	39
3.0 Directives.....	41
3.1 #chip.....	41
3.2 #config.....	41
3.3 #define.....	41
3.4 #if.....	42
3.5 #ifdef.....	43
3.6 #ifndef.....	45
3.7 #include.....	45
3.8 #script.....	46
3.9 #startup.....	46
4.0 Advanced Features.....	47
4.1 Arrays.....	47
4.2 Functions.....	48
4.3 Interrupts.....	48
4.4 Lookup Tables.....	49

4.5 Scripts	50
4.6 Subroutines	51
4.7 PS2 Keypad Overview	55
4.8 Random Overview	56
4.9 I2C Overview	57
4.10 Timer Overview	58
4.11 Seven Segment Overview	58
4.12 PWM Overview	60
4.13 Software RS232 Overview	62
4.14 Hardware RS232 Overview	63
4.15 Keypad Overview	64
4.16 LCD Overview	66
4.17 Tone Overview	70
4.18 SPI Overview	71
4.19 GLCD Overview	72
5.0 Commands	75
Box	75
ClearTimer	75
CLS	76
Dim	77
Dir	79
DisplayChar	80
DisplayValue	81
Do	83
End	85
EPRead	85
EPWrite	87
Exit Sub	89
FilledBox	90
For	90
Get	92
GLCDCLS	92
GLCDPrint	93
GLCDDrawChar	93
GLCDWriteByte	94
GLCDReadByte	94
Gosub	94
Goto	96
HPWM	97
HSerPrint	99
HSerReceive	104
HSerSend	105
I2CReceive	107
I2CSend	109
I2CStart	111
I2CStop	112

If Then Else	112
InitGLCD	114
InitSer	116
InitTimer0	117
InitTimer1	120
INKEY	122
Interrupts	125
IntOff	126
IntOn	127
KeypadData	127
KeypadRaw	130
LCDCreateChar	132
LCDHex	134
LCDWriteChar	135
Line	137
Locate	137
On Interrupt	138
Peek	143
Poke	144
Pot	145
Print	147
ProgramErase	148
ProgramRead	149
ProgramWrite	150
PS2ReadByte	150
PS2SetKBLeds	151
PS2WriteByte	152
PSet	153
PulseOut	153
Put	154
PWMOff	156
PWMOn	157
PWMOut	158
Random	159
Randomize	160
ReadAD	161
ReadTable	162
Repeat	164
Rotate	165
Select	167
SerPrint	170
SerReceive	175
SerSend	176
Set	177
ShortTone	179
SPIMode	180

SPITransfer	181
StartTimer	184
StopTimer	185
Tone	185
Wait.....	186
6.0 Sample Projects.....	190
Project 1 - Train Crossing	190
Project 2 - Sensing a Switch	194
Project 3 – Creating Sound	197
Project 4 - Sensing Light.....	200
Appendix A – PICKIT 2 GUI Features	204
UART Tool	204
Connecting the UART Tool.....	205
UART Tool Window	206
Setting the Baud Rate and Connecting	206
ASCII Mode.....	207
Hex Mode.....	208
Wrap Text	208
Log to File.....	208
Clear Screen.....	209
Exit UART Tool	209
Logic Tool.....	209
Logic I/O Mode.....	210
Configuring the Logic Tool Logic I/O.....	210
Setting Pin Direction.....	211
Logic Analyzer Mode	213
Connecting the Analyzer.....	213
The Logic Analyzer Window.....	214
Analyzer Display	215
Analyzer Display Cursors	216
Analyzer Trigger.....	217
Analyzer Acquisition	218
Running the Analyzer.....	220

1.0 Introduction

Great Cow BASIC (GCB) is a BASIC compiler for PIC microcontrollers. It lets you to program in BASIC instead of having to learn assembly or C language. It's also completely open source!

CHIPINO is an open source development module that makes programming with Great Cow Basic, Assembly or C language much easier. It has the same footprint as the popular Arduino modules so all the various plug in shields that work with Arduino can work with CHIPINO.

CHIPINO uses a programming cable based on the open source PICKIT 2 programmer from Microchip. This allows the CHIPINO to be used with any blank 28 pin PIC microcontroller. This also allows the user to unplug the chip and build it into a permanent design when the software is complete.

This makes the combination of Great Cow Basic and CHIPINO a perfect development platform for creating electronic products.

Great Cow BASIC has been written with three main aims - to remove the need for repetitive assembly commands, to produce efficient code, and to make it easy to take code written for one chip and run it on another. It hides many of the more confusing parts of microcontroller programming, making it suitable for beginners and those who don't like assembly or C. It's also great for kids to learn electronic programming. And aren't we all just kids inside?

The syntax of Great Cow BASIC is based on that of QBASIC/FreeBASIC, but with some alterations to suit the vastly different system that makes up a microcontroller. Great Cow BASIC will allow you to program most 8 bit PIC microcontrollers (PIC10F, 12F, 16F and 18F chips).

Features:

- Standard BASIC flow control statements - If, Select Case, Do, For, Goto
- Support for multiply, divide, add, subtract, boolean operations and comparisons.
- Bit, Byte, Word and String data types, in addition to byte arrays.
- Subroutines and Functions
- Inline assembly, in most cases without any special directives
- Data tables

Portable, reusable code:

- Supports most 8-bit PIC microcontrollers
- Write code for a PIC10F, 12F , 16F or 18F 8-bit PIC Microcontroller

- Automatically recalculates all delay commands depending on the clock speed of the chip

I/O capabilities:

- Standard 2x16 LCD routines
- Routines for on-chip A/D, PWM, SPI, USART, EEPROM and Timers
- RS232 communications - rates between 300 and 19200 bps with user configurable parity, start and stop bits.
- PS/2 keyboard reading
- 4x4 Keypad

Other features:

- Extensive help file and forum
- Several translations
- Generates standard MPASM compatible assembly code for PIC.
- Free, both in the free beer way and the free speech way!
- Open Source.

Great Cow Basic is created by a group of key contributors including:

Program Contributors:

Hugh Considine - Main developer of Great Cow Basic

Stefano Bonomi - Two-wire LCD subroutines

Geordie Millar - Swap and Swap4 subroutines

Finn Stokes - 8-bit multiply routine, program memory access code

Translation Contributors (We only support English in this manual):

Stefano Delfiore - Italian

Pablo Curvelo - Spanish

Murat Inceer - Turkish

Other Contributors:

Russ Hensel - Great Cow BASIC Notes

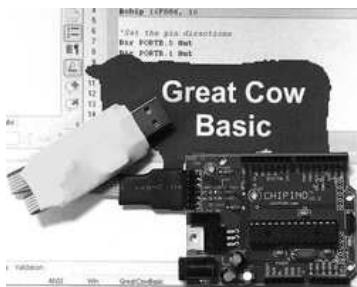
Chuck Hellebuyck - This manual.

Frank Steinberg - GCB@SYN IDE for Great Cow Basic

Alexy T. - SynWrite IDE used for GCB IDE

1.1 Using Great Cow Basic

The best way to start using Great Cow Basic is with a complete starter kit. You can get the CHIPINO module, Great Cow Basic software for Windows, USB Programmer cable, CHIPINO module, sample software projects and this ebook manual in pdf form in one complete starter package.



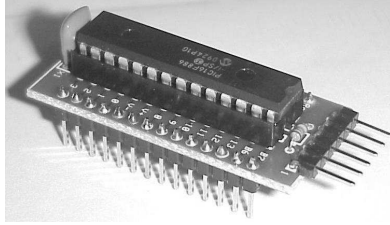
With this setup all you need is a PC running Windows, XP, Vista or Win7 or a Mac running Parallels or VMWare and Windows.

The CHIPINO makes it easy for a beginner to use Great Cow Basic because of all the pre-built plug-in shields that stack on top. There are numerous code examples for this starter kit at www.greatcowbasic.com.

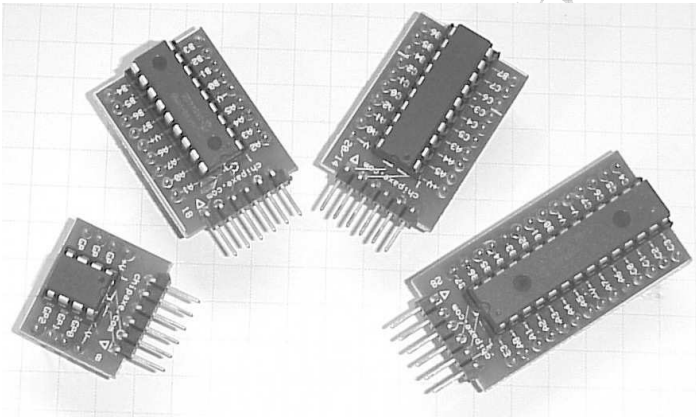


LCD Shield and Demo Shield

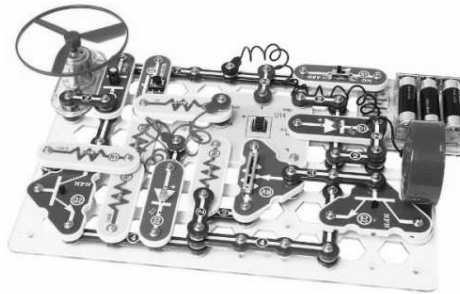
You can also use it with a CHIPINO Breadboard Module.



You can also use Great Cow Basic with the CHIPAXE modules from Howtronic.com. These modules make it easy to use GCB with 8 pin, 14 pin, 18 pin, 20 pin and 28 pin Microchip PICs.



You can also use Great Cow Basic to program an 8 pin PIC Microcontroller and use it with Snap Circuits®. There are also adapters to plug a CHIPAXE module into a Snap Circuits® module.



Minimum Required

All you really need to use Great Cow Basic is a Microchip PIC Microcontroller and a programmer. It works great with a Microchip PICKit2 or one of the many PICKit 2 clones since the PICKit 2 design is also open source.



Original PICKit 2



PK2 Clone



PICKit 2 Clone

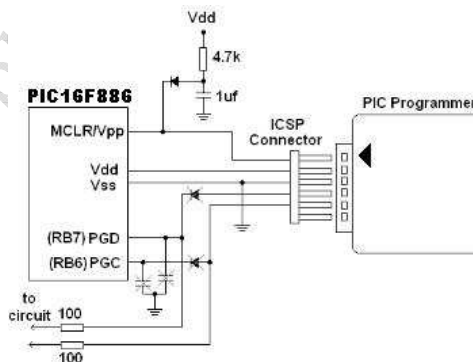
The Great Cow Basic Compiler supports all the 8-bit Microchip PICs including all the parts that start with PIC18F, PIC16F, PIC12F and PIC10F. A very common part to start with is the PIC16F886. It's a 28 pin part that has most of the features all the other PICs have including:

- Internal Oscillator
- Internal MCLR resistor option
- 10 bit Analog to Digital Converter (ADC)
- 22 I/O
- EEPROM
- UART
- SPI / I2C hardware
- Timers
- And more

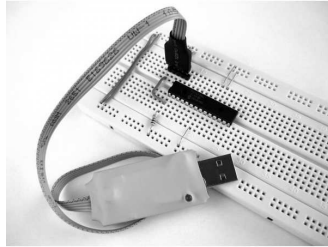


PIC16F886

The Connections to the programmer are shown in the picture below. The CHIPINO modules take care of all these connections for you but you can build it your self in a breadboard as shown.



In-Circuit Serial Programming (ICSP)



PIC in a Breadboard

1.2 Using Great Cow Basic

The Great Cow Basic compiler can be downloaded from the greatcowbasic.com website as a complete software package. Installation instructions are below.

Installation

Requirements:

Windows PC running XP, or higher (or Mac running Parallels/Windows, Linux running Wine)

USB Port

There are two ways to install Great Cow Basic CHIPINO (GCBC) on your PC; Automatic Installation or Manual Installation.

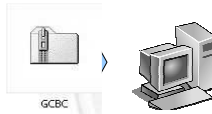
Automatic Installation

- 1) Download the GCBC Setup file and run it on your computer. It will install GCBC in your start menu and also place an icon on your desktop.



Manual Installation

- 1) If available you can also download the GCBCIDEVx.zip file from the greatcowbasic.com website.



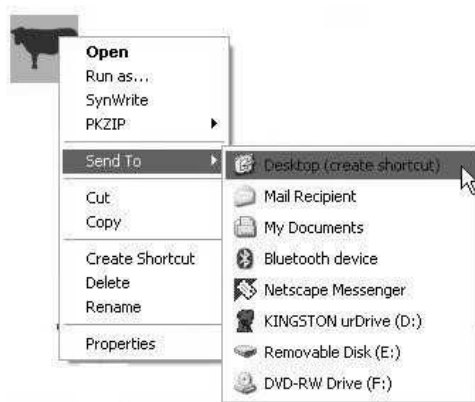
- 2) Unzip it to a directory on your computer.



- 3) Open the GCBCIDEVx folder and you'll see several folders.



- 4) Right click on the Cow symbol to create a shortcut on your desktop.



5) You should now have a GCBC icon on your desk top.



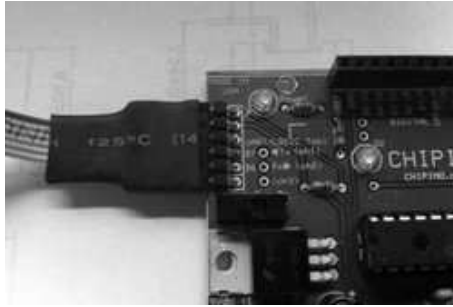
6) Installation is complete. You are ready to create your first project.

1.3 Creating your first project

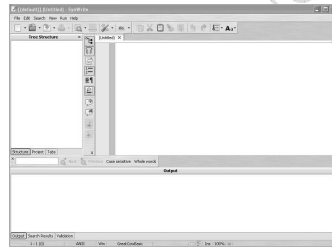
1) Connect the Programming Cable to an open USB port.



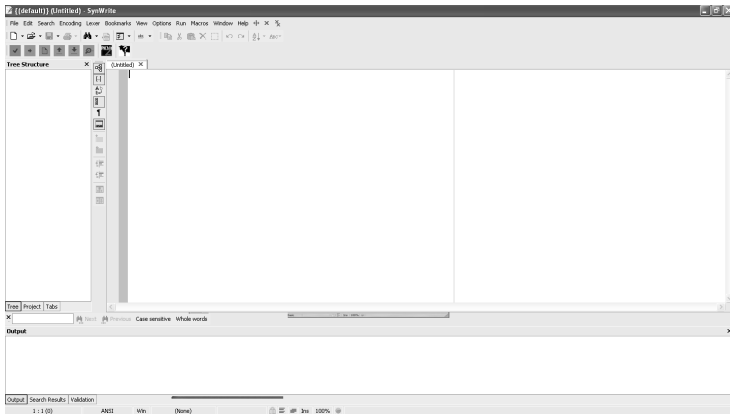
2) Connect the programming cable to the CHIPINO. Make sure the red line on the cable lines up with the \wedge arrow (by the power LED) on the CHIPINO.



3) Double click on the GCBC icon to launch the GCBC IDE.



4) You'll see a blank screen the first time you open the IDE.



At the top you'll see the GCBC Toolbar for creating software projects.



They represent the following functions:



Compile the GCB file and report any errors (This will create a .hex file).



Compile and then program the CHIPINO through the PICKit2 Programming Cable.



Open a new blank window.



Open existing GCB file.



Save the GCB file.



Open a Terminal File for communication through the computer

serial port. This requires a separate RS232 shield and RS232 port on your PC or a RS232 to USB converter cable.




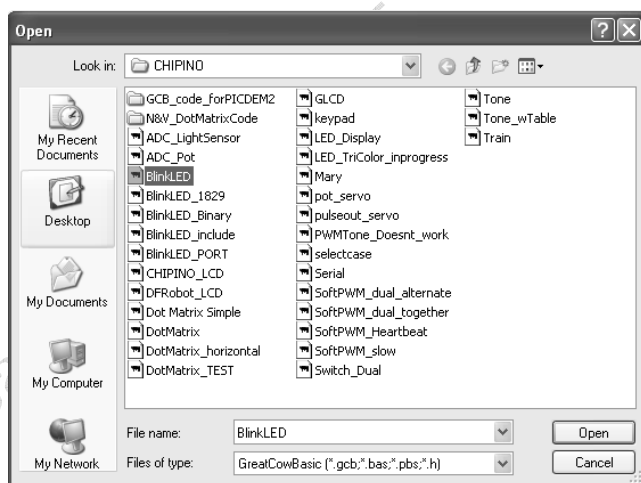
Open the PICKIT 2 Standalone GUI software so you can use the PICKIT 2 Programming cable with the built in Terminal or Logic Analyzer. See Appendix A for more detail on these features.



Open the GCB help file with details on the GCB commands and features.

5) Now open the file BlinkLED.GCB from the CHIPINO folder.

Click on  and then select the BlinkLED.gcb file from the CHIPINO folder. Click on Open button.




6) You should see the BlinkLED.gcb program in the editor window.

```

BlinkLED.gcb* x
1      'A program to flash LED on PIN 13/PORTB5
2
3      'Chip model
4      #chip 16F886, 16
5
6      'Main routine
7      Start:
8          'Turn one LED on, the other off
9          SET PORTB.5 ON
10         Wait 1 sec
11         'Now toggle the LEDs
12         SET PORTB.5 OFF
13         Wait 1 sec
14         'Jump back to the start of the program
15         Goto Start
16
17

```

7) Click on  or just press the F5 key. The Great Cow Basic Compiler will convert the BlinkLED.gcb file into the 1's and 0's the microcontroller, within the CHIPINO module, needs to run. A window will pop up showing the compile and programming of the CHIPINO was successful. The LED will start to blink on the CHIPINO.

```

CoSteel V20130114 www.franksteinberg.de -->co13--
*** sub000K.bas ***
Great Cow BASIC (0.9 8/11/2012)
Compiling C:\GCB IDE V9\CHIPINO\BlinkLED.gcb ...
Done
Assembling program ...
Program assembled successfully!
*** flash32C.bas ***
PICkit 2 Program Report
00-Feb-2013, 16:44:15
Device Type: PIC16F886
Program Succeeded.
Operation Succeeded

```



```

Program Succeeded.
Operation Succeeded

```

You just successfully programmed the CHIPINO for the first time.

1.4 Command Line Options

If you are more of the technical type of user who wants to create your programs in text editor and then program via command line rather than through the IDE then GCB at the heart is a command line compiler. Included here are the options for this type of programming.

GCBASIC [/O:output.asm] [/A:assembler] [/P:programmer] [/K:{C|A}] [/V] [/L]/[NP] filename

Switch	Description	Default
/O:filename	Sets the name of the assembly file generated to <i>filename</i> .	Same name as the input file, but with a .asm extension.
/A:assembler	Batch file used to call assembler*. If /A:GCASM is given, GCBASIC will use its internal assembler.	The program will not be assembled
/P:programmer	Batch file used to call programmer*. This parameter is ignored if the program is not assembled.	The program will not be downloaded.
/K:{C A}	Keep original code in assembly output. /K:C will save comments, /K:A will preserve all input code.	No original code left in output.
/V	Verbose mode - compiler gives more detailed information about its activities.	None
/L	Show license and exit.	None
/NP	Do not pause on errors. Use with IDEs.	Pause when an error occurs, and wait for the user to press a key.
filename	The file to compile.	None

* For the /A: and /P: switches, there are special options available.

If %FILENAME% is present, it will be replaced by the name of the .asm file.

If %FN_NOEXT% will be replaced by the name of the .asm file but without an extension and %CHIPMODEL% will be replaced with the name of the chip. The name of the chip will be the same as that on the chip data file.

2.0 Compiler Basics

2.1 Inputs/Outputs

Most general purpose pins on a microcontroller can function in one of two modes: input mode, or output mode.

When acting as an input, the pin will be placed in high impedance state. The microcontroller will then sense the pin, and the program can read the state of the pin and make decisions based on it.

When in output mode, the microcontroller will connect the pin to either Vcc (the positive supply), or Vss (ground, or the negative supply) based on the commands in the GCB program. There are various commands that can set the state of the pin.

GCB will attempt to determine the direction of each pin, and set it appropriately. However, if the pin is both read from and written to during the program, then the pin must be configured to input / output mode by the program, using the appropriate Dir command.

Syntax:

Dir *port.bit* {In | Out} (**Individual Form**)

Dir *port* {In | Out | *DirectionByte*} (**Entire Port Form**)

Explanation:

The Dir command is used to set the direction of the ports of the microcontroller chip. The individual form sets the direction of one pin at a time, whereas the entire port form will set all bits in a port. This controls the TRIS register within the PIC if you are familiar with the inner workings.

In the individual form, specify the port and bit (ie. PORTB.4), then the direction, which is either In or Out.

The entire port form is similar to the TRIS instruction offered by some PIC chips. To use it, give the name of the port (ie. PORTA), and then a byte is to be written into the TRIS variable. *This form of the command is for those who are familiar with the PIC chip's internal architecture.*

WARNING: PICs use 0 for out and 1 for in. When IN and OUT are used there are no compatibility issues.

Example:

'This program sets PORTA bits 0 and 1 to in, and the rest to out.

'It also sets all of PORTB to output, except for B1.

'Individual form is used for PORTA:

DIR PORTA.0 IN

DIR PORTA.1 IN

DIR PORTA.2 OUT

DIR PORTA.3 OUT

DIR PORTA.4 OUT

DIR PORTA.5 OUT

DIR PORTA.6 OUT

DIR PORTA.7 OUT

'Entire port form used for B:

DIR PORTB b'00000010'

'Entire port form used for C:

DIR PORTC IN

2.2 Configuration

Every PIC chip has a CONFIG word. This is an area of memory on the chip that stores settings which govern the operation of the chip.

The following aspects of the chip are governed by the CONFIG word:

- Oscillator selection - will the chip run from an internal oscillator, or is an external one attached?
- Automatic resets - should the chip reset if the power drops too low? If it detects it is running the same piece of code over and over?
- Code protection - what areas of memory must be kept hidden once written to?
- Pin usage - which pins are available for programming, resetting the chip, or emitting PWM signals?

The exact configuration settings vary amongst chips. To find out a list of valid settings, please consult the datasheet for the PIC chip that you wish to use.

This can all be rather confusing - hence, GCB will automatically set some config settings, unless told otherwise:

- Low Voltage Programming (LVP) is turned off.** This enables the PGM pin (usually B3 or B4) to be used as a normal I/O pin.

•**Watchdog Timer (WDT) is turned off.** The WDT resets the chip if it runs the same piece of code over and over - this can cause trouble with some of the longer delay routines in GCBASIC.

•**Master Clear (MCLR) is disabled where possible.** On many newer chips this allows the MCLR pin (often PORTA.5) to be used as a standard input port. It also removes the need for a pull-up resistor on the MCLR pin.

•**An oscillator mode will be selected, based on the following rules in this order of priority:**

1. If the PIC has an internal oscillator, and the internal oscillator is capable of generating the speed specified in the #chip line, then the internal oscillator will be used.
2. If the clock speed is over 4 Mhz, the external HS oscillator is selected
3. If the clock speed is 4 MHz or less, then the external XT oscillator mode is selected.

Note that these settings can be individually overridden whenever needed. For example, if the Watchdog Timer is needed, adding the line below will enable the watchdog timer, without affecting any other configuration settings.

```
#config WDT = ON
```

Using Configuration

Once the necessary CONFIG options have been determined, adding them to the program is easy. On a new line type "#config" and then list the desired options separated by commas, such as in this line:

```
#config OSC = RC, BODEN = OFF
```

GCB also supports this format on 10/12/16 series chips:

```
#config INTOSC_OSC_NOCLKOUT, BODEN_OFF
```

However, for upwards compatibility with 18F chips, you should use the first config settings option.

It is possible to have several #config lines in a program - for instance, one in the main program, and one in each of several #include files. However, care must then be taken to ensure that the settings in one file do not conflict with those in another.

2.3 Variables

A variable is an area of memory on the microcontroller that can be used to store a number or a series of letters. This is useful for many purposes, such as taking a sensor reading and acting on it, or counting the number of times the robot has performed a particular task.

Each variable must be given a name, such as "MyVariable" or "PieCounter". Choosing a name for a variable is easy - just don't include spaces or any symbols (other than `_`), and make sure that the name is at least 2 characters (letters and/or numbers) long.

Variable Types

There are several different types of variable, and each type can store a different sort of information. These are the variable types that Great Cow BASIC can currently use:

Variable type	Information that this variable can store	Example uses for this type of variable
Bit	A bit (0 or 1)	Flags to track whether or not a piece of code has run
Byte	A whole number between 0 and 255	General purpose storage of data, such as counters
Word	A whole number between 0 and 65535	Storage of extra large numbers
Integer	A whole number between -32768 and 32767	Anything where a negative number will occur
Long	A whole number between 0 and 2^{32} (4.29 billion)	Storing very, very big numbers
Array	A list of whole numbers ranging from 0 to 255	Logs of sensor readings
String	A series of letters, numbers and symbols.	Messages that are to be shown on a screen

Using Variables

Byte variables are automatically created in Great Cow Basic. Just put the name of the variable in to the command where the variable is needed and the compiler will create the variable automatically.

Other types of variable must be "dimensioned" first. This involves using the DIM command, to tell Great Cow BASIC that it is dealing with something other than a byte variable.

Dim Var as Word 'Variable VAR can now hold 0 to 65,535
Dim Flag as Bit 'Flag variable will hold a 0 or 1

A key feature of variables is that it is possible to have the microcontroller check a variable, and only run a section of code if it is a given value. This can be done with the IF command.

Variable Aliases

Some variables are aliases, which are used to refer to memory locations used by other variables. These are useful for joining predefined byte variable together to form word variables.

Aliases are not like pointers in many languages - they must always refer to the same variable or variables and cannot be changed.

One useful Alias is the HighByte, Low Byte for word variables.

Example:

Dim BigVar As Word Alias HighByte, LowByte

HighByte = 2
LowByte = 100

You can also access bits of a variable like this. Remember though that the first bit is bit 0 not bit 1. So all eight bits of a byte variable are number 0 thru 7.

If Var.2 = 1 then ' Test bit 3 of variable Var to see if its set to 1
Set D13 on ' Light LED on D13 if it's set
End If

Setting Variables

Syntax:

Variable = data

Explanation:

Variable will be set to *data*. *Data* can be either a fixed value (such as 157), another variable, or a sum.

If *data* is a fixed value, it must be an integer between 0 and 255 inclusive.

If *data* is a calculation, then it may have any of the following operands:

+ (add)

- (subtract, or negate if there is no value before it)

* (multiply)

/ (divide)

% (modulo)

& (and)

| (or)

(xor)

! (not)

= (equal)

<> (not equal)

< (less than)

> (greater than)

<= (less than or equal)

>= (more than or equal)

The final 6 operands are for checking conditions. They will return FALSE (0) if the condition is false, or TRUE (255) if the condition is true.

The And, Or, Xor and Not operators function both as bitwise and logical operators.

GCBASIC understands order of operations. If multiple operands are present, they will be processed in this order:

1. Brackets
2. Unary operations (not and negate)
3. Multiply/Divide/Modulo
4. Add/Subtract
5. Conditional operators
6. And/Or/Xor

There are several modes in which variables can be set. GCBASIC will automatically use a different mode for each calculation, depending on the type of variable being set. If a byte variable is being set, byte mode will be used; if a word variable is being set, word mode will be used. If a byte is being set but the calculation involves numbers larger than 255, word mode can be used by adding [WORD] to the start of one of the values in the calculation. This is known as casting.

If you prefer, you can add "LET" to the start of the line. It will not alter the execution of the program, but is included for those who are used to including it in other BASIC dialects.

Example:

This program is to illustrate the setting of variables.

Chipmunk = 46 'Sets the variable Chipmunk to 46

Animal = Chipmunk 'Sets the variable Animal to the value of the variable Chipmunk

Bear = 2 + 3 * 5 'Sets the variable Bear to the result of 2 + 3 * 5, 17.

Sheep = (2 + 3) * 5 'Sets the variable Sheep to the result of (2 + 3) * 5, 25.

Animal = 2 * Bear 'Sets the variable Animal to twice the value of Bear.

LargeVar = 321 'LargeVar must be set as a word - see DIM.

Temp = LargeVar / [WORD]5 'Note the use of [WORD] to ensure that the calculation is performed correctly

2.4 Casting

Casting changes the type of a variable or value. Placing the type that the value should be converted to in square brackets will tell the compiler to convert it. For example, this will cause two byte variables to be treated as word variables by the addition code:

```
Dim MyWord As Word
MyWord = [word]ByteVar + AnotherByteVar
```

Why do this? If there are no casts, then GCBASIC will add the two values using the byte addition code, and then convert the result to a word to store in MyWord. Suppose that ByteVar is 150, and AnotherByteVar is 231. When added, this will come to 381 - which will overflow, leaving 125 in the result. However, when the cast is added, GCBASIC will treat ByteVar as if it were a word, and so will use the word addition code. This will cause the correct result to be calculated.

Often, a cast will be used when calculating an average:

```
MyAverage = ([word]Value1 + Value2) / 2
```

It's also possible to cast the second value:

```
MyAverage = (Value1 + [word]Value2) / 2
```

The result will be exactly the same.

2.5 Defines/Constants

Syntax:

```
#define Find Replace
```

Explanation:

`#define` will search through the program for *Find*, and replace it with the value given for *Replace*.

A define or constant is a type of directive that tells the compiler to find a given word, and replace it with another word or number. Defines are useful for situations where a routine needs to be easily altered. For example, a define could be used to specify the amount of time to run an alarm for once triggered.

It is also possible to use defines to specify ports - thus defines can be used to aid in the creation of code that can easily be adapted to run on a different PIC with different ports.

GCBASIC makes considerable use of defines internally. For instance, the LCD code uses defines to set the ports that it must use to communicate with the LCD.

Using Defines

To create a define is a matter of using the `#define` directive. Here are some examples of defines:

```
#define Line 34
```

Line is a simple constant - GCBASIC will find "Line" in the program, and replace it with the number 34. This could be used in a line following program, to make it easier to calibrate the program for different lighting conditions.

```
#define Light PORTB.0
```

Light is a port - it represents a particular pin on the PIC chip. This would be of use if the program had many lines of code that controlled the light, and there was a possibility that the port the light was attached to would need to change in the future.

```
#define LightOn Set PORTB.0 on
```

LightOn is a define used to make the program more readable. Rather than typing "Set PORTB.0 on" over and over, it would then be made possible to type "LightOn", and have the compiler do the hard work.

2.6 Numeric Constants

GCB allows numeric constants to be defined in three bases: decimal, binary and hexadecimal. Decimal are the default and require no prefix. Binary and Hex numbers can be written in a couple formats but each requires a prefix.

Decimal:

170

Binary:

0b10100101 ' Binary equivalent to 170
or
b'10100101'

Hex:

H'A5' ' Hex equivalent to 170
or
0xA5

2.6.1 BCD to Decimal

Sometimes you need to format numbers in Binary Coded Decimal format. GCB doesn't have the function built in but you can easily add it with this function below. Just add this to your GCB program and then call it when you need it.

```
Function DecToBcd(va) as Byte
DecToBcd=(va/10)*16+va%10
End Function
```

Example:

BCDNumber = DecToBcd(number) 'Convert variable number to BCD value

2.6.2 Decimal to BCD

Sometimes you need to convert numbers from Binary Coded Decimal format to decimal. GCB doesn't have the function built in but you can easily add it with this function below. Just add this to your GCB program and then call it when you need it.

```
Function BcdToDec(va) as byte
BcdToDec=(va/16)*10+va%16
End Function
```

Example:

DecNumber = BcdToDec(number) 'Convert variable number to Decimal value

2.7 String Constants

String variables are useful for many communication commands such as send to an LCD. Strings are characters placed between quotes.

If a variable is set equal to a string, the ASCII value equivalent will be used.

"A" will read as the ASCII value of 65

"d" will read as the ASCII value of 100

Strings can be combined in a single phrase within quotes.

For example:

The string "Hello" is the same as "H", "e", "l", "l", "o"

Here is how strings can be useful for sending letters to an LED display.

```
'This program will show "Hello" on a LED display
'The display is connected to PORTB
```

```
Dim Message As string
Message = "Hello "
For Counter = 1 to 6
    DisplayChar 1, Message(Counter)
    Wait 250 ms
Next
```

You can also combine strings like variables.

```
Dim TestString$ as string
DIM OtherString$ as string
```

```
TestString$ = "Hello"           'Contains Hello
OtherString$ = TestString$ + " World" 'OtherString$ contains Hello World
```

```
'On LCD Screen
Print TestString$ 'Print Hello
Print OtherString$ 'Print Hello World
```

2.8 String Conversion

These commands make it easy to convert String Constants and Variables to other formats.

2.8.1 String to Hex

The Hex function will convert a number into hexadecimal format. The input *number* should be a byte variable, or a fixed number between 0 and 255 inclusive. After running the function, the string variable *stringvar* will contain a 2 digit hexadecimal number.

Syntax:

stringvar = Hex(*number*)

Example:

```
'Uses Hex to display as hexadecimal
For CurrentLocation = 0 to 255
    HSerPrint Hex(CurrentLocation)
Next
```

2.8.2 String to Decimal Value

The Val function will extract a number from a string variable, and store it in a word variable. One potential use is reading numbers that are sent in ASCII format over a serial connection.

Syntax:

var = Val(*string*)

Example:

```
'Variables for received bytes
Dim DataIn As String

'Convert output level to numeric variable
OutputLevel = Val(DataIn)

'Output
HPWM 1, 32, OutputLevel
```

2.8.3 Find String within a String

The Instr function will search one string to find the location of another string within it. *source* is the string to search inside, and *find* is the string to find. The function will return the location of *find* within *source*, or 0 if *source* does not contain *find*.

Syntax:

location = Instr(source, find)

Example:

```
Dim TestData As String
TestData = "Hello, world!"

'Display the location of "world" within the string
'Will return 8, because "w" in world is the 8th character
'of "Hello, world!"
HSerPrint Instr(TestData, "world")
HSerPrintCRLF

'Display the location of "planet" within the string
'Will display 0, because "planet" does not occur inside
'the string "Hello, world!"
HSerPrint Instr(TestData, "planet")
HSerPrintCRLF
```

2.8.4 Convert to Lower Case

The LCase function will convert all of the letters in the string *source* to lower case, and return the result.

Syntax:

output = LCase(*source*)

Example:

```
'Set chip model
#chip 16F1936

'Set up hardware serial connection
#define USART_BAUD_RATE 9600
#define USART_BLOCKING

'Fill a string with a message
Dim TestData As String
TestData = "Hello, world!"

'Display the string in lower case
'Will display "hello, world!"
HSerPrint LCase(TestData)
HSerPrintCRLF
```

2.8.5 Convert to Upper Case

The UCase function will convert all of the letters in the string *source* to upper case, and return the result.

Syntax:

```
output = UCase(source)
```

Example:

```
Dim TestData As String
TestData = "Hello, world!"

'Display the string in upper case
'Will display "HELLO, WORLD!"
HSerPrint UCase(TestData)
```

2.8.6 Convert Number to String

The Str function will convert a number into a string. *number* can be any byte or word variable, or a fixed number between 0 and 65535 inclusive. The string variable *stringvar* will contain the same number, represented as a string.

This function is especially useful if a number needs to be added to the end of a string, or if a custom data sending routine has been created but only supports the output of string variables.

Syntax:

```
stringvar = Str(number)
```

Example:

```
'Take an A/D reading
SensorReading = ReadAD(AN0)

'Create a string variable
Dim OutVar As String

'Fill string with sensor reading
OutVar = Str(SensorReading)

'Send
HSerPrint OutVar
HSerPrintCRLF
```

2.8.7 Retrieve String Characters (Left)

The Left function will extract the leftmost *count* characters from the input string *source*, and return them in a new string.

Syntax:

```
output = Left(source, count)
```

Example:

```
Dim TestData As String
TestData = "Hello, world!"
```

```
'Display the leftmost 5 characters
'Will display "Hello"
HSerPrint Left(TestData, 5)
```

2.8.8 Retrieve String Characters (Middle)

The Mid function is used to extract characters from the middle of a string variable. *source* is the variable to extract from, *start* is the position of the first character to extract, and *count* is the number of characters to extract. If *count* is not specified, all characters from *start* to the end of the source string will be returned.

Syntax:

output = Mid(*source*, *start*, *count*)

Example:

```
'Extract "cat". The c is at position 5, and 3 letters are needed
HSerPrint "The animal is a "
HSerPrint Mid(TestData, 5, 3)
```

```
'Extract the action. "sat" starts at position 9.
HSerPrint "The animal "
HSerPrint Mid(TestData, 9)
HSerPrintCRLF
```

2.8.9 Retrieve String Characters (Right)

The Right function will extract the rightmost *count* characters from the input string *source*, and return them in a new string.

Syntax:

output = Right(*source*, *count*)

Example:

```
Dim TestData As String
TestData = "Hello, world!"

'Display the rightmost 6 characters
'Will display "world!"
HSerPrint Right(TestData, 6)
HSerPrintCRLF
```

2.9 Mathematical Operations

If *data* is a calculation, then it may have any of the following operands:

+ (add)
- (subtract, or negate if there is no value before it)
***** (multiply)
/ (divide)
% (modulo)
++ (increment)
-- (decrement)

Bitwise and logical operators:

& (and)
| (or)
(xor)
! (not)
= (equal)
<> (not equal)
< (less than)
> (greater than)
<= (less than or equal)
>= (more than or equal)

The final 6 operands are for checking conditions. They will return FALSE (0) if the condition is false, or TRUE (255) if the condition is true.

The And, Or, Xor and Not operators function both as bitwise and logical operators.

GCBASIC understands order of operations. If multiple operands are present, they will be processed in this order:

1. Brackets
2. Unary operations (not and negate)
3. Multiply/Divide/Modulo
4. Add/Subtract
5. Conditional operators
6. And/Or/Xor

There are several modes in which variables can be set. GCBASIC will automatically use a different mode for each calculation, depending on the type of variable being set. If a byte variable is being set, byte mode will be used; if a word variable is being set, word mode will be used.

If a byte is being set but the calculation involves numbers larger than 255, word mode can be used by adding [WORD] to the start of one of the values in the calculation. This is known as casting (section 2.4).

If you prefer, you can add "LET" to the start of the line. It will not alter the execution of the program, but is included for those who are used to including it in other BASIC dialects.

Example:

'This program is to illustrate the setting of variables.

Chipmunk = 46 'Sets the variable Chipmunk to 46

Animal = Chipmunk 'Sets the variable Animal to the value of Chipmunk

Bear = 2 + 3 * 5 'Sets the variable Bear to the result of 2 + 3 * 5, 17

Sheep = (2 + 3) * 5 'Sets the variable Sheep to the result of (2 + 3) * 5, 25

Animal = 2 * Bear 'Sets the variable Animal to twice the value of Bear

LargeVar = 321 'LargeVar must be set as a word - see DIM

Temp = LargeVar / [WORD]5 'LargeVar is a word size variable.

'Note the use of [WORD] to ensure that the calculation is performed correctly

2.10 Conditions

In GCBASIC (and most other programming languages) a condition is a statement that can be either true or false. Conditions are used when the program must make a decision.

A condition is generally given as a value or variable, a relative operator (such as = or >), or another value or variable. Several conditions can be combined to form one condition through the use of logical operators such as AND and OR.

GCBASIC supports these relative operators:

Symbol	Meaning
=	Equal
<>	Not Equal
<	Less Than

>	Greater Than
<=	Less than or equal to
>=	Equal to or greater than

In addition, these logical operators can be used to combine several conditions into one:

Name	Abbreviation	Condition true if
AND	&	both conditions are true
OR		at least one condition is true
XOR	#	one condition is true
NOT	!	the condition is not true

NOT is slightly different than the other logical operators, in that it only needs one other condition. Other arithmetic operators may be combined in conditions, to change values before they are compared.

GCBASIC has two built in conditions - TRUE, which is always true, and FALSE, which is always false. These can be used to create infinite loops.

It is also possible to test individual bits in conditions. To do this, specify the bit to test, then 1 or 0 (or ON and OFF respectively). Presently there is no way to combine bit tests with other conditions - NOT, AND, OR and XOR will not work.

Example conditions:

Condition	Comments
Temp = 0	Condition is true if Temp = 0
Sensor <> 0	Condition is true if Sensor is not 0
Reading1 > Reading2	True if Reading1 is more than Reading2
Mode = 1 AND Time > 10	True if Mode is 1 and Time is more than 10
Heat > 5 OR Smoke > 2	True if Heat is more than 5 or Smoke is more than 2
Light >= 10 AND (NOT Time > 7)	True if Light is 10 or more, and Time is 7 or less
Temp.0 ON	True if Temp bit 0 is on

2.11 Combining Multiple Instructions

It is possible to combine multiple instructions on a single line, by separating them with a colon. For example, this code:

```

Set PORTB.0 On
Set PORTB.1 On
Wait 1 sec
Set PORTB.0 Off
Set PORTB.0 Off

```

Could also be written as:

```

Set PORTB.0 On: Set PORTB.1 On
Wait 1 sec
Set PORTB.0 Off: Set PORTB.0 Off

```

In most cases, it will make no difference if commands share a line or not. However, special care should be taken with "IF" commands, as this code:

```

Set PORTB.0 Off
Set PORTB.1 Off
If Temp > 10 Then Set PORTB.0 On: Set PORTB.1 On
Wait 1 s

```

Will be equivalent to this:

```

Set PORTB.0 Off
Set PORTB.1 Off
If Temp > 10 Then
    Set PORTB.0 On
    Set PORTB.1 On
End If
Wait 1 s

```

Also, the commands used to start and end subroutines, data tables and functions must be alone on a line. For example, this is WRONG:

```

Sub Something: Set PORTB.0 Off: End Sub

```

2.12 Comments

Adding comments to your GCB program is done with an apostrophe before the comment line. You can also comment out sections of code if you want just by placing an apostrophe at the beginning of each line. The SynGCB IDE has a feature to do this automatically. It will comment out or comment in blocks of code that you highlight. There is a comment out and comment in icon in the center toolbar.



Comment and Uncomment Icons

In some GCB sample programs you may also see comment lines starting with a semi-colon (;) or the term REM.

```
;chip settings
Rem This is a comment line
```

Neither of these are automatically recognized by the IDE auto formatting so they will look like command lines but the Great Cow Basic compiler will recognize them as commented lines and not include them in the final compile.

It's best to just use the apostrophe for all commented lines so the IDE will show them as non highlighted comment lines. This will make your code easier to read. The comment/uncomment toolbar icons will add apostrophes or remove them.

Example

```
'The number of pins to flash
#define FlashPins 2

'You can create a header using an apostrophe before each line
'This is a great way to describe your program
'You can also use it to describe the hardware connections.

'You can place comments above the command or on the same line
Dir PORTB Out 'Initialise PORTB to all Outputs

'Main loop
do
    PORTB = 0      'All Pins off
    Wait 1 S       'Delay 1 second
    PORTB = 0xFF   'All pins on
    Wait 1 s       'Delay 1 second
Loop
```

3.0 Directives

3.1 #chip

Syntax:

#chip model, speed

Explanation:

The *#chip* directive is used to specify the chip model and speed that GCBASIC must compile for. *model* is the model of the microcontroller chip - something along the lines of "16F819". *speed* is the speed of the chip in MHz, and is required for the delay and PWM routines.

If *speed* is omitted, GCBASIC will use the highest clock speed that the internal oscillator can support. If the chip does not have an internal oscillator, then GCBASIC will assume that the chip is being run at its maximum possible clock speed using an external crystal.

Examples:

```
#chip 12F509, 4
#chip 18F4550, 48
#chip 16F88, 0.125
#chip tiny2313, 1
#chip mega8, 16
```

3.2 #config

Syntax:

#config option1, option2, ... , optionn

Explanation:

The *#config* directive is used to specify configuration options for the chip. There is a detailed explanation of *#config* in the Configuration section of help.

3.3 #define

Syntax:

#define Find Replace

Explanation:

#define will search through the program for *Find*, and replace it with the value given for *Replace*.

3.4 #if**Syntax:**

```
#if Condition
...
#endif
```

Explanation:

The #if directive is used to prevent a section of code from compiling unless *Condition* is true.

Condition has the same syntax as the condition in a normal GCBASIC if command. The only difference is that it uses constants instead of variables.

Example:

```
'This program will pulse an adjustable number of pins on PORTB
'The number of pins is controlled by the FlashPins constant
#chip 16F88, 8
```

```
'The number of pins to flash
#define FlashPins 2
```

```
'Initialise
Dir PORTB Out
```

```
'Main loop
Do
  #if FlashPins >= 1
    PulseOut PORTB.0, 250 ms
  #endif
  #if FlashPins >= 2
    PulseOut PORTB.1, 250 ms
  #endif
  #if FlashPins >= 3
    PulseOut PORTB.2, 250 ms
  #endif
  #if FlashPins >= 4
    PulseOut PORTB.3, 250 ms
  #endif
Loop
```

3.5 #ifdef

Syntax:

```
#ifdef Constant | Constant Value | Var(VariableName)
...
#endif
```

Explanation:

The #ifdef directive is used to selectively enable sections of code. There are several ways in which it can be used:

- Checking if a constant is defined
- Checking if a constant is defined and has a particular value
- Checking if a system variable exists
- Checking if a system bit has been defined

The advantage of using #ifdef rather than an equivalent series of IF statements is the amount of code that is downloaded to the chip. #ifdef controls what code is compiled and downloaded, IF controls what is run once on the chip. #ifdef should be used whenever the value of a constant is to be checked.

GCBASIC also supports the #ifndef directive - this is the opposite of the #ifdef directive - it will remove code that #ifdef leaves, and vice versa.

(Note: The code in the following sections will not compile, as it is missing #chip directives and DIR commands. It is intended to act as an example only.)

Enabling code if a constant is defined

Syntax Example:

```
#define Blink1

#ifdef Blink1
    PulseOut PORTB.0, 1 sec
    Wait 1 sec
#endif
#ifdef Blink2
    PulseOut PORTB.1, 1 sec
    Wait 1 sec
#endif
```

This code will pulse PORTB.0, but not PORTB.1. This is because Blink1 has been defined, but Blink2 has not. If the line

```
#define Blink2
```

was added at the start of the program, then both pins would be pulsed. The value of the constant defined is not important and can be left off of the #define.

Enabling code if a constant is defined and has a given value

Syntax Example:

```
#define PinsToFlash 2

#ifdef PinsToFlash 1,2,3
    PulseOut PORTB.0, 1 sec
#endif
#ifdef PinsToFlash 2,3
    PulseOut PORTB.1, 1 sec
#endif
#ifdef PinsToFlash 3
    PulseOut PORTB.2, 1 sec
#endif
```

This program uses a constant called PinsToFlash that controls how many lights are pulsed. PORTB.0 is pulsed when PinsToFlash is equal to 1, 2 or 3, PORTB.1 is pulsed when PinsToFlash equals 2 or 3, and PORTB.2 is flashed when PinsToFlash is 3.

Enabling code if a system variable is defined

Syntax Example:

```
#ifdef NoVar(ANSEL)
    SET ADCON1.PCFG3 OFF
    SET ADCON1.PCFG2 ON
    SET ADCON1.PCFG1 ON
    SET ADCON1.PCFG0 OFF
#endif
#ifdef Var(ANSEL)
    ANSEL = 0
#endif
```

The above section of code has been copied directly from a-d.h. It is used to disable the A/D function of pins, so that they can be used as standard digital I/O ports. If ANSEL is not declared as a system variable for a particular chip, then the program uses ADCON1 to control the port modes. If ANSEL is defined, then the chip is newer and its ports can be set to digital by clearing ANSEL.

Enabling code if a system bit is defined

Similar to above, except with Bit and NoBit in the place of Var and NoVar respectively.

3.6 #ifndef

Syntax:

```
#ifndef Constant | Constant Value | Var(VariableName)  
...  
#endif
```

Explanation:

The #ifndef directive is used to selectively enable sections of code. It is the opposite of the #ifdef directive - it will delete code in cases where #ifdef would leave it, and will leave code where #ifdef would delete it.

3.7 #include

Syntax:

```
#include filename
```

Explanation:

#include tells GCBASIC to open up another file, read all of the subroutines and constants from it, and then copy them into the current program.

There are two forms of include - absolute, and relative.

Absolute is used to refer to files in the C:\Program Files\GCBASIC\include directory. The name of the file is specified in between < and > symbols. For instance, to include the file "srf04.h", the directive is:

```
#include <srf04.h>
```

Relative is used to read files in the same folder as the currently selected program. Filenames are given enclosed in quotation marks, such as:

```
#include "mycode.h"
```

where mycode.h is the name of the file that is to be read.

It is not essential that the include file name ends in .h - the important thing is that the name given to GCBASIC is the exact name of the file to be included.

Those who are familiar with #include in assembly or C should bear in mind that #include in GCBASIC works differently to #include in most other languages -

code is not inserted at the location of the #include, but rather at the end of the current program.

If an appropriate File Converter is installed, then #include can be used to include non-GCBASIC files. GCBASIC will detect that the file extension of the included file matches a converter, and run the appropriate converter to import the file.

3.8 #script

Syntax:

```
#script
    [scriptcommand1]
    [scriptcommand2]
    ...
    [scriptcommandn]
#endscript
```

Explanation:

The #script block is used to create small sections of code which GCBASIC runs during compilation. A detail explanation and example are included in the Scripts article.

3.9 #startup

Syntax:

```
#startup SubName
```

Explanation:

#startup is used to include files to automatically insert initialization routines. If a define or subroutine from the file is used in the program, then the specified subroutine will be called.

There are some limitations on this directive. It may only occur once within a file, and no parameters can be specified for the subroutine that is to be called.

4.0 Advanced Features

4.1 Arrays

An array is a special type of variable - one which can store several values at once. It is essentially a list of numbers in which each one can be addressed individually through the use of an "index". The index is a value in brackets immediately after the name of the array.

Examples of array names are:

Array/Index	Meaning
Fish(10)	Element 10 of an array called Fish
DataLog(2)	The second number in an array named DataLog
ButtonList(Temp)	An element in the array "ButtonList" that is selected according to the value in the variable "Temp"

Using Arrays

To use an array, its name is specified, then the index. Arrays can be used everywhere that a normal variable can be used.

Setting an entire array at once

It is possible to set several elements of an array with a single line of code. This short example shows how:

```
Dim TestVar(10)
TestVar = 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Element 0 of TestVar will be set to the number of items in the list, which in this case is 9. Each element of the array will then be loaded with the corresponding value in the list - so in the example, TestVar(1) will be set to 1, TestVar(2) to 2, and so on.

If there are many items in the array, it may be better to use a Lookup Table to store the items, and then copy some of the data items into a smaller array as needed.

4.2 Functions

Functions are a special type of subroutine that can return a value. This means that when the name of the function is used in the place of a variable, GCBASIC will call the function, get a value from it, and then put the value into the line of code in the place of the variable.

Functions may also have parameters - these are treated in exactly the same way as parameters for subroutines. The only exception is that brackets are required around any parameters when calling a function.

Using Functions

This program uses a function called AverageAD to take two analog readings, and then make a decision based on the average:

```
'Select chip
#chip 16F88, 20

'Define ports
#define LED PORTB.0
#define Sensor AN0

'Set port directions
dir LED out
dir PORTA.0 in

'Main code
Do
    Set PORTB.0 Off
    If AverageAD > 128 Then Set PORTB.0 On
    wait 10 ms
Loop

Function AverageAD
    'Get 2 readings, divide by 2, store in AverageAD
    'Note the cast, the result of ReadAD needs to be converted to
    'a word before adding, or the result may overflow.
    AverageAD = ([word]ReadAD(Sensor) + ReadAD(Sensor)) / 2
end function
```

4.3 Interrupts

Interrupts are a feature of many microcontrollers. They allow the microcontroller to temporarily pause (interrupt) the code it is running and then start running

another piece of code when some event occurs. Once it has dealt with the event, it will return to where it was and continue running the program.

Many events can trigger an interrupt, such as a timer reaching its limit, a serial message being received, or a special pin on the microcontroller receiving a signal.

Using Interrupts

There are two ways to use interrupts in GCBASIC. The first way is to use the On Interrupt command. This will automatically enable a given interrupt, and run a particular subroutine when the interrupt occurs.

The other way to deal with interrupts is to create a subroutine called Interrupt. GCBASIC will call this subroutine whenever an interrupt occurs, and then your code can check the "flag" bits to determine which interrupt has occurred, and what should be done about it. If you use this approach, then you'll need to enable the desired interrupts manually. It is also essential that your code clears the flag bits, or else the interrupt routine will be called repeatedly.

Some combination of these two methods is also possible - the code generated by On Interrupt with check to see if the interrupt is one it recognises. If the interrupt is recognised, On Interrupt will deal with it - if not, the Interrupt subroutine will be called to deal with the interrupt.

The recommended way is to use On Interrupt, as it is both more efficient and easier to set up.

During some sections of code, it is desirable not to have any interrupts occur. If this is the case, then use the IntOff command to disable interrupts at the start of the section, and IntOn to re-enable them at the end. If any interrupt events occur while interrupts are disabled, then they will be processed as soon as interrupts are re-enabled. If the program does not use interrupts, IntOn and IntOff will be removed automatically by GCBASIC.

4.4 Lookup Tables

A lookup table is a list of values that are stored in the program memory of the chip, which can be accessed using the ReadTable command.

The advantage of lookup tables is that they are memory efficient, compared to an equivalent set of IF statements.

Using Lookup Tables

First, the table must be created. The code to create a lookup table is simple - a line that has "Table" and then the name of the table, a list of numbers (up to 254), and then "End Table".

Once the table is created, the ReadTable command is used to read data from it. The ReadTable command requires the name of the table it is to read, the location of the item to retrieve, and a variable to store the retrieved number in.

Lookup tables can store byte or word values. GCBASIC will automatically detect the type of the table depending on the values in it.

Item 0 of a lookup table stores the size of the table. If the ReadTable command attempts to read beyond the end of the table, the value 0 will be returned.

4.5 Scripts

A script is a small section of code that Great Cow BASIC runs when it starts to compile a program. Their main use is to perform calculations that are required to adjust the program for different speed chips.

Scripts are not compiled and downloaded to the microcontroller - GCBASIC reads them, runs them, removes them from the program and then allows any results they have calculated to be used as constants in the program.

Inside a script, constants are treated like variables. Scripts can read the values of constants, and set them to contain new values.

Using Scripts

Scripts start with "#script" and end with "#endscript". Inside, they can consist of 3 commands:

- If
- Assignment (=)
- Error

If is similar to the If command in normal GCBASIC code, except that it doesn't have an Else clause. It is used to compare the values of constants.

= is identical to that in GCBASIC programs. The constant that is to be set goes on the left side of the =, and the new value goes on the right.

Error is used to display an error message. Anything after the Error command is displayed at the end of compilation, and is saved in the error log for the program.

Example Script

This script is used in the pwm.h file. It takes the values of the constants PWM_Freq, PWM_Duty and ChipMHz, and uses the equations shown in the PIC datasheets to calculate the correct values for the relevant system variables.

```
#script
PR2Temp = int((1/PWM_Freq)/(4*(1/(ChipMHz*1000))))
T2PR = 1
If PR2Temp > 255 Then
    PR2Temp = Int((1 / PWM_Freq) / (16 * (1 / (ChipMHz * 1000))))
    T2PR = 4
If PR2Temp > 255 Then
    PR2Temp = Int((1 / PWM_Freq) / (64 * (1 / (ChipMHz * 1000))))
    T2PR = 16
    If PR2Temp > 255 Then
        Error Invalid PWM Frequency value
    End If
End If
End If

DutyCycle = (PWM_Duty * 10.24) * PR2Temp / 1024
DutyCycleH = (DutyCycle AND 1020) / 4
DutyCycleL = DutyCycle AND 3
#endscript
```

After this script has run, the values PR2Temp, DutyCycleH and DutyCycleL are used as constants to set up the required variables.

4.6 Subroutines

A subroutine is a small program inside of the main program. Subroutines are typically used when a task needs to be repeated several times in different parts of the main program.

There are two main uses for subroutines:

- Keeping programs neat and easy to read
- Reducing the size of programs by allowing common sections of code to be reused.

When the PIC comes to a subroutine it saves its location in the current program before jumping to the start of, or calling, the subroutine. Once it reaches the end of the subroutine it returns to the main program, and continues to run the code where it left off previously.

Normally, it is possible for subroutines to call other subroutines. There are limits to the number of times that a subroutine can call another sub, which vary from chip to chip:

PIC Family	Instruction Width	Number of subs called
10F*, 12C5*, 12F5*, 16C5*, 16F5*	12	1
12C*, 12F*, 16C*, 16F*, except those above	14	7
18F*, 18C*	16	31

These limits are due to the amount of memory on the PIC which saves its location before it jumps to a new subroutine. Some GCBASIC commands are subroutines, so you should always allow for 2 or 3 subroutine calls more than your program has.

Another feature of subroutines is that they are able to accept parameters. These are values that are passed from the main program to the subroutine when it is called, and then passed back when the subroutine ends.

Using Subroutines

To call a subroutine is very simple - all that is needed is the name of the sub, and then a list of parameters. This code will call a subroutine named "Buzz" that has no parameters:

```
Buzz
```

If the sub has parameters, then they should be listed after the name of the subroutine. This would be the command to call a subroutine named "MoveArm" that has three parameters:

```
MoveArm NewX, NewY, 10
```

If a subroutine has parameters, you may choose to put brackets around them, like so:

```
MoveArm (NewX, NewY, 10)
```

All that this does is change the appearance of the code - it doesn't make any difference to what the code does. Decide which one meets your own personal preference, and then stick with it.

Creating subroutines

To create a subroutine is almost as simple as using one. There must be a line at the start which has "sub ", and then the name of the subroutine. Also, there needs to be a line at the end of the subroutine which reads "end sub". To create a subroutine called "Buzz", this is the required code:

```
sub Buzz
```

```
'code for the subroutine goes here
```

```
end sub
```

If the subroutine has parameters, then they need to be listed after the name. For example, to define the "MoveArm" sub used above, use this code:

```
sub MoveArm(ArmX, ArmY, ArmZ)
```

```
'code for the subroutine goes here
```

```
end sub
```

In the above sub, ArmX, ArmY and ArmZ are all variables. If the call from above is used, the variables will have these values at the start of the subroutine:

```
ArmX = NewX
```

```
ArmY = NewY
```

```
ArmZ = 10
```

When the subroutine has finished running, GCBASIC will copy the values back where possible. NewX will be set to ArmX, and NewY to ArmY. GCBASIC will not attempt to set the number 10 to ArmZ.

Controlling the direction data moves in

It is possible to instruct GCBASIC not to copy the value back after the subroutine is called. If a subroutine is defined like this:

```
sub MoveArm(In ArmX, In ArmY, In ArmZ)
```

```
'code for the subroutine goes here
```

```
end sub
```

Then GCBASIC will copy the values to the subroutine, but will not copy them back.

GCBASIC can also be prevented from copying the values back, by adding "Out" before the parameter name. This is used in the EEPROM reading routines - there is no point copying a data value into the read subroutine, so Out has been used to avoid wasting time and memory. The EPRead routine is defined as "Sub EPRead(In Address, Out Data).

Many older sections of code use "#NR" at the end of the line where the parameters are specified. The "#NR" means "No Return", and when used has the same effect as adding "In" before every parameter. Use of "#NR" is not recommended, as it does not give the same level of control.

Using different data types for parameters

It is possible to use any type of variable as a parameter for a subroutine. Just add "As " and then the data type to the end of the parameter name. For example, to make all of the parameters for the MoveArm subroutine word variables, use this code:

```
sub MoveArm(ArmX As Word, ArmY As Word, ArmZ As Word)
...
end sub
```

Optional parameters

Sometimes, the same value may be used over and over again for a parameter, except in a particular case. If this occurs, a default value may be specified for the parameter, and then a value for that parameter only needs to be given in a call if it is different to the default.

For example, suppose a subroutine to create an error beep is required. Normally it emits a 440 Hz tone, but sometimes a different tone is required. To create the sub, this code would be used:

```
Sub ErrorBeep(Optional OutTone As Word = 440)
    Tone OutTone, 100
End Sub
```

Note the "Optional" before the parameter, and the " = 440" after it. This tells GCBASIC that if no parameter is supplied, then set the OutTone parameter to 440.

If called using this line:

```
ErrorBeep
```

then a 440 Hz beep will be emitted. If called using this line:

ErrorBeep 1000

then the sub will produce a 1000 Hz tone.

When using several parameters, it is possible to make any number of them optional. If the optional parameter/s are at the end of the call, then no value needs to be specified. If they are at the start or in the middle, then you must insert commas to allow GCBASIC to tell where the optional parameters are.

Overloading

It is possible to have 2 subroutines with the same name, but different parameters. This is known as overloading, and GCBASIC will automatically select the most appropriate subroutine for each call.

An example of this is the Print routine in the LCD routines. There are actually 3 Print subroutines; one has a byte parameter, one a word parameter, and one a string parameter. If this command is used:

Print 100

Then the Print (byte) subroutine will be called. However, if this command is used:

Print 30112

Then the Print (word) subroutine will be called. If there is no exact match for a particular call, GCBASIC will use the option that requires the least conversion of variable types. For example, if this command is used:

Print PORTB.0

The byte print will be used. This is because byte is the closest type to the single bit parameter.

4.7 PS2 Keypad Overview

Introduction

These routines make it easier to communicate with a PS/2 device, particularly a keyboard.

Relevant Constants

These constants affect the operation of the PS/2 routines:

Constant Name	Controls	Default Value
PS2Data	Pin connected to PS/2 data line	N/A
PS2Clock	Pin connected to PS/2 clock line.	N/A
PS2_DELAY	This constant can be set to a delay, such as 10 ms. If set, a delay will be added at the end of every byte sent or received.	Not set

4.8 Random Overview

Introduction:

These routines allow GCBASIC to generate pseudo-random numbers.

The generator uses a 16 bit linear feedback shift register to produce pseudo-random numbers. The most significant 8 bits of the LFSR are used to provide an 8 bit random number.

When compiling a program, GCBASIC will generate an initial seed for the generator. However, this seed will be the same every time the program runs, so the sequence of numbers produced by a given program will always be the same. To work around this, there is a Randomize subroutine. It can be provided with a new seed for the generator (which will cause the generator to move to a different point in the sequence). Alternatively, Randomize can be set to obtain a seed from some other source such as a timer every time it is run.

Relevant Constants:

These constants are used to control settings for the tone generation routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name	Controls	Default Value
RANDOMIZE_SEED	Source of the random seed if Randomize is called without a parameter	Timer0

4.9 I2C Overview

Introduction:

These routines allow GCBASIC programs to send and receive I2C messages. They can be configured to act as master or slave, and the speed can also be altered.

No hardware I2C module is required for these routines - all communication is handled in software. However, these routines will not work on 12-bit instruction PICs (10F, 12F5xx and 16F5xx chips).

Relevant Constants:

These constants control the setup of the software I2C routines.

Constant	Controls	Default Value
I2C_MODE	Mode of I2C routines (Master or Slave)	Master
I2C_DATA	Pin on microcontroller connected to I2C data	N/A
I2C_CLOCK	Pin on microcontroller connected to I2C clock	N/A
I2C_BIT_DELAY	Time for a bit (used for acknowledge detection)	2 us
I2C_CLOCK_DELAY	Time for clock pulse to	1 us

	remain high	
I2C_END_DELAY	Time between clock pulses	1 us
I2C_USE_TIMEOUT	Set to true if slave mode I2C routines should stop waiting for the master and exit after a timeout occurs.	Not Set

4.10 Timer Overview

Several functions are provided to read the current timer value. They are:

- Timer0
- Timer1
- Timer2
- Timer3
- Timer4
- Timer5

Not all of these functions are available on all chips. For example, if a chip only has 3 timers, then only Timer0, Timer1 and Timer2 will be available. Timer0 and Timer2 return byte values, while Timer1, Timer3, Timer4 and Timer5 will return words.

Please refer to the datasheet for your microcontroller to determine the number and size of the timers available.

4.11 Seven Segment Overview

Introduction

The 7 segment display routines make it easier for GCBASIC programs to display numbers and letters on 7 segment LED displays.

There are two ways that the 7 segment display routines can be set up. One option is to connect the wires from the display/s in a particular order, and then to set the DisplayPort n and DispSelect n constants. The other option is to connect the display/s in whatever way is easiest, and then set the DISP_SEG_x and DISP_SEL_x constants. The first option (setting DisplayPort x and DispSelect n) will generate slightly more efficient code.

Configuration using DISP_SEG_x and DISP_SEL_x

When setting up the 7 segment code with DISP_SEG_x constants, these must be set:

Constant Name	Controls	Default Value
DISP_SEG_x	Controls the output pin used to control segment x of the displays. There are 7 of these constants, named DISP_SEG_A through DISP_SEG_G. One must be set for each segment.	N/A
DISP_SEG_DOT	Specifies the output pin used to control the decimal point on the displays.	N/A
DISP_SEL_x	The command used to select display n . Used to control addressing pins when several displays are multiplexed.	N/A

Note: Instead of setting DISP_SEL_x, it is possible to set DispSelect n and use these in conjunction with DISP_SEG_x.

Configuration using DisplayPort*n* and DispSelect*n*

To set the 7-Segment display routines supplied with GCBASIC using DisplayPort*n*, it is necessary to set these constants:

Constant Name	Controls	Default Value
DisplayPort <i>n</i>	Controls the output port used to control display <i>n</i> . <i>n</i> is A, B, C or D, corresponding to displays 1, 2, 3 and 4, respectively.	N/A
DispSelect <i>n</i>	The command used to select display <i>n</i> . Used to control addressing pins when several displays are multiplexed.	nop

To set up the routines in this way, the displays must be connected as follows:

Microcontroller port pin	Display Segment
0	A
1	B
2	C
3	D
4	E
5	F
6	G

4.12 PWM Overview

Introduction:

The routines described in this chapter allow the generation of Pulse Width Modulation signals. These allow for the microcontroller to control the speed of a

motor, or the brightness of a light. The routines can also be used to generate the appropriate frequency signal to drive an infrared LED for remote control applications.

The PWMOn, PWMOFF and HPWM routines use the PWM generation module on the microcontroller. They will only work on some microcontrollers - see the article on each command for details. PWMOn and HPWM will cause the PWM module on the microcontroller to start generating the PWM signal, which will then continue to be generated until the PWMOFF instruction is run.

PWMOut does not make use of any special hardware. However, a signal is only generated while the PWMOut command is running - when the program moves on to the next command, the signal will stop.

Relevant Constants:

These constants are used to control settings for the Pulse Width Modulation module of the PIC chip. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Note that there are two sets of constants: one for Hardware PWM, and one for Software PWM. Hardware PWM requires a CCP module on the PIC chip - Software PWM has no requirements regarding the PIC.

Hardware PWM

These constants are only required for PWMOn. HPWM and PWMOFF do not require any constants to operate.

Constant Name	Controls	Default Value
PWM_Freq	Specifies the output frequency of the PWM module in KHz.	38
PWM_Duty	Sets the duty cycle of the PWM module output. Given as percentage.	50

Hardware PWM is only available through the "CCP1" or "CCP" pin. This is a hardware limitation of PIC microcontrollers.

Software PWM

Constant Name	Controls	Default Value
PWM_Delay	The PWM Period. The length of any delay used will be multiplied by 255. If no value is specified, no delays will be inserted into the PWM routine.	Not defined - no delay
PWM_Out <i>n</i>	The port physical port on the PIC that corresponds to channel <i>n</i> . <i>n</i> can represent 1, 2, 3 or 4.	Not Defined

More than 4 channels are possible, but for this the PWMOut routine in `include\lowlevel\stdbasic.h` must be altered.

4.13 Software RS232 Overview

Introduction:

These routines allow the microcontroller to send and receive RS232 data.

All functions are implemented using software, so no special hardware is required on the microcontroller. However, if the microcontroller has a hardware serial module (usually referred to as UART or USART), and the serial data lines are connected to the appropriate pins, the hardware routines should be used for smaller code, improved reliability and higher baud rates.

Relevant Constants:

These constants are used to control settings for the RS232 serial communication routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name/s	Controls	Default Value
SendALow, SendBLow, SendCLow	These are used to define the commands used to send a low (0) bit on serial channels A, B and C respectively.	nop
		(must be set)
SendAHigh, SendBHigh, SendCHigh	These are used to define the commands used to send a high (1) bit on serial channels A, B and C respectively.	nop
		(must be set)
RecALow, RecBLow, RecCLow	The condition that is true when a low bit is being received	Sys232Temp.0 OFF
		(must be set)
RecAHigh, RecBHigh, RecCHigh	The condition that is true when a high bit is being received	Sys232Temp.0 ON
		(must be set)

4.14 Hardware RS232 Overview

Introduction

These subroutines allow GCBASIC programs to communicate more easily using RS232.

These hardware-based routines are intended for use on microcontrollers with built in RS232 modules - normally referred to in datasheets as USART or UART modules. To use these, the RS232 data lines must be connected to the pins on the microcontroller used by the serial module. If the RS232 lines are connected elsewhere, or the microcontroller has no RS232 module, then the software based routines must be used.

Relevant Constants

These constants affect the operation of the hardware RS232 routines:

Constant Name	Controls	Default Value
USART_BAUD_RATE	Baud rate (in bps) for the routines to operate at.	N/A
USART_BLOCKING	If set, this constant will cause the USART routines to delay until data can be sent or received. If not set, then the data will be buffered and send by the hardware when possible.	Not set

4.15 Keypad Overview

Introduction

The keypad routines allow for a program to read from a 4 x 4 matrix keypad.

There are two ways that the keypad routines can be set up. One option is to connect the wires from the keypad in a particular order, and then to set the KeypadPort constant. The other option is to connect the keypad in whatever way is easiest, and then set the KEYPAD_ROW_x and KEYPAD_COL_x constants. The first option (setting KeypadPort) will generate slightly more efficient code.

Configuration using KEYPAD_ROW_x and KEYPAD_COL_x

These constants must be set:

Constant Name	Controls	Default Value
KEYPAD_ROW_1	The pin on the microcontroller that connects to the Row 1 pin on the keypad	N/A
KEYPAD_ROW_2	The pin on the microcontroller that connects to the Row 2	N/A

	pin on the keypad	
KEYPAD_ROW_3	The pin on the microcontroller that connects to the Row 3 pin on the keypad	N/A
KEYPAD_ROW_4	The pin on the microcontroller that connects to the Row 4 pin on the keypad	N/A
KEYPAD_COL_1	The pin on the microcontroller that connects to the Col 1 pin on the keypad	N/A
KEYPAD_COL_2	The pin on the microcontroller that connects to the Col 2 pin on the keypad	N/A
KEYPAD_COL_3	The pin on the microcontroller that connects to the Col 3 pin on the keypad	N/A
KEYPAD_COL_4	The pin on the microcontroller that connects to the Col 4 pin on the keypad	N/A

If using a 3 x 3 keypad, do not set the KEYPAD_ROW_4 or KEYPAD_COL_4 constants.

Configuration using KeypadPort

When setting up the keypad code using the KeypadPort constant, only KeypadPort needs to be set:

Constant Name	Controls	Default Value
KeypadPort	The port on the microcontroller chip that the keypad is connected to.	N/A

For this to work, the keypad must be connected as follows:

Microcontroller port pin	Keypad connector
0	Row 1
1	Row 2
2	Row 3
3	Row 4
4	Column 1
5	Column 2
6	Column 3
7	Column 4

Note: To use a 3 x 3 keypad in this mode, the pins on the microcontroller for any unused columns must be pulled up.

4.16 LCD Overview

Introduction:

The routines in this section allow GCBASIC programs to interact with alphanumeric Liquid Crystal Displays based on the HD44780 IC. This covers most 16 x 2 and similar displays.

These routines allow the displays to be connected to the microcontroller in many different ways:

Connection Mode	Required Connections
0	None are required directly by the routines. However, the LCD routines must be provided with other subroutines which will handle the communication. This is useful for communicating with LCDs connected through RS232 or I2C.
2	Data and Clock lines. This mode is used when the LCD is connected through a 74LS174 shift register IC, as detailed at http://www.rentron.com/Myke1.htm .

4	R/W, RS, Enable and the highest 4 data lines (DB4 through DB7).
8	R/W, RS, Enable and all 8 data lines. The data lines must all be connected to the same I/O port, in sequential order. For example, DB0 to PORTB.0, DB1 to PORTB.1 and so on, with DB7 going to PORTB.7.

Using 0-bit mode:

To use 0-bit connection mode, a subroutine to write a byte to the LCD must be provided. Optionally, another subroutine to read a byte from the LCD can also be given. If there is no way to read from the LCD, then the LCD_NO_RW constant must be set.

In 0-bit mode, the LCD_RS constant will be set automatically to a spare bit variable. The higher level LCD commands (such as Print and Locate) will set it, and the code responsible for writing to the LCD should read it and then set the RS pin on the LCD appropriately.

This code is an example of how to use 0-bit mode. It sends messages to another microcontroller, which has been programmed to read the messages and toggle the pins of an LCD appropriately:

```
#define LCD_IO 0

#define LCDWriteByte MySendToLCD

#define LCD_NO_RW

Sub MySendToLCD(In MyLCDByte)

    'Uses I2C

    'Sends an address byte (128)

    'Then a control byte, where bit 4 is the state of the RS pin

    'Then a data byte, which is sent to the LCD data pins.
```

```

ControlByte = 0

If LCD_RS = On Then ControlByte.4 = On

I2CStart

I2CSend 128

I2CSend ControlByte

I2CSend MyLCDByte

I2CStop

'Need to allow time for receiver of message to update LCD

Wait 5 ms

End Sub

```

If the LCD was to be read, then LCDReadByte would be set to the name of a function that reads the LCD and returns the data byte from the LCD.

Relevant Constants:

(Note: this section does not apply in 0 bit mode)

These constants are used to control settings for the Liquid Crystal Display routines included with GCBASIC. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Some constants are used required for 4 and 8-bit modes, some are required for 4-bit mode, and some are used by 8-bit mode. When using 2-bit mode only three constants must be set - all others can be ignored. Check the "Modes" column to determine if you must set a constant.

Constant Name	Controls	Default Value	Modes
LCD_IO	The I/O mode. Can be 2, 4 or 8.	8	2, 4, 8
LCD_DB	The data pin used in 2-bit mode.	N/A - Must be set	2
LCD_CB	The clock pin used in 2-bit mode.	N/A - Must be set	2
LCD_RS	Specifies the output pin that is connected to Register Select on the LCD.	N/A - Must be set	4, 8
LCD_RW	Specifies the output pin that is connected to Read/Write on the LCD. The R/W pin can be disabled*.	N/A - Must be set (unless R/W is disabled)	4, 8
LCD_Enable	Specifies the output pin that is connected to Read/Write on the LCD.	N/A - Must be set	4, 8
LCD_DATA_PORT	Output port used to interface with LCD data bus	N/A - Must be set	8 only
LCD_DB4	Output pin used to interface with bit 4 of the LCD data bus	N/A - Must be set	4 only
LCD_DB5	Output pin used to interface with bit 5 of the LCD data bus	N/A - Must be set	4 only
LCD_DB6	Output pin used to interface with bit 6 of the LCD data bus	N/A - Must be set	4 only
LCD_DB7	Output pin used to interface with bit 7 of the LCD data bus	N/A - Must be set	4 only

*The R/W pin can be disabled by setting the LCD_NO_RW constant. If this is done, there is no need for the R/W to be connected to the chip, and no need for the LCD_RW constant to be set. Ensure that the R/W line on the LCD is connected to ground if not used!

4.17 Tone Overview

Introduction:

These routines generate tones of a given frequency and duration.

Note: If an exact frequency is required, or a smaller program is needed, these routines should not be used. Instead, you should use code like this:

```
Repeat count
```

```
    PulseOut SoundOut, period us
```

```
    Wait period us
```

```
End Repeat
```

Set *count* and *period* to the appropriate values:

- *period* to $1000000 / \text{desired frequency} / 2$
- *count* to $\text{desired duration} / \text{period}$.

Relevant Constants:

These constants are used to control settings for the tone generation routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name	Controls	Default Value
SoundOut	The output pin to produce sound on	N/A - Must be set

4.18 SPI Overview

Command Availability:

Available on PIC microcontrollers with Synchronous Serial Port (SSP) module. These commands control the SPI communication.

Syntax:

SPIMode Mode

SPIMode sets the mode of the SPI module within the PIC chip. These are the possible SPI Modes:

Mode Name	Description
MasterSlow	Master mode, SPI clock is 1/64 of the frequency of the PIC.
Master	Master mode, SPI clock is 1/16 of the frequency of the PIC.
MasterFast	Master mode, SPI clock is 1/4 of the frequency of the PIC.
Slave	Slave mode
SlaveSS	Slave mode, with the Slave Select pin enabled.

Syntax:

SPITransfer tx, rx

This command simultaneously sends and receives a byte of data using the SPI protocol. It behaves differently depending on whether the PIC has been set to act as a master or a slave.

When operating as a master, the SPITransfer command will initiate a transfer. The data in tx will be sent to the slave, whilst the byte that is buffered in the slave

will be read into *rx*.

In slave mode, the SPITransfer command will pause the program until a transfer is initiated by the master. At this point, it will send the data in *tx* whilst reading the transmission from the master into the *rx* variable.

4.19 GLCD Overview

The GLCD commands are used to control a graphical Liquid Crystal Display based on the KS0108 driver chip. These are often 128x64 pixel displays. They can draw graphical elements by enabling or disabling pixels.

GCB makes this easier to control with the following commands.

Setup:

You must include the `glcd.h` file at the top of your program. The file needs to be in brackets.

```
#include <GLCD.h>
```

Defines:

There are several connections that must be defined to use the GLCD commands with a KS0108 display. The *I/O pin* is the pin on the PIC Microcontroller that is connected to that specific pin on the graphic LCD.

```
#define GLCD_RW I/O pin    'Read/Write pin connection
#define GLCD_RESET I/O pin 'Reset pin connection
#define GLCD_CS1 I/O pin   'CS1 pin connection
#define GLCD_CS2 I/O pin   'CS2 pin connection
#define GLCD_RS I/O pin    'RS pin connection
#define GLCD_ENABLE I/O pin 'Enable pin Connection
#define GLCD_DB0 I/O pin   'Data pin 0 Connection
#define GLCD_DB1 I/O pin   'Data pin 1 Connection
#define GLCD_DB2 I/O pin   'Data pin 2 Connection
#define GLCD_DB3 I/O pin   'Data pin 3 Connection
#define GLCD_DB4 I/O pin   'Data pin 4 Connection
#define GLCD_DB5 I/O pin   'Data pin 5 Connection
#define GLCD_DB6 I/O pin   'Data pin 6 Connection
#define GLCD_DB7 I/O pin   'Data pin 7 Connection
```

Commands

InitGLCD 'Initialize port pins

GLCDCLS 'Clear screen

GLCDPrint(In PrintLocX, In PrintLocY, PrintData As String)

GLCDDrawChar(In CharLocX, In CharLocY, In CharCode)

Box(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

FilledBox(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

Line(In LineX1, In LineY1, In LineX2, In LineY2, Optional In LineColour = 1)

PSet(In GLCDX, In GLCDY, In GLCDState)

GLCDWriteByte (In LCDByte)

GLCDReadByte

Sample Version - Do Not Copy

5.0 Commands

Box

Syntax:

Box(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a box on a graphic LCD from the upper corner of X1, Y1 location to X2,Y2 location.

See also FilledBox command

ClearTimer

Syntax:

ClearTimer *TimerNo*

Explanation:

ClearTimer is used to clear the specified timer.

Example:

Please refer to the InitTimer1 article for an example.

CLS

Syntax:

CLS

Explanation:

The CLS command clears the contents of the LCD, and returns the cursor to the top left corner of the screen

Example:

```
'A Flashing Hello World program for GCBASIC
```

```
'General hardware configuration
```

```
#chip 16F877A, 20
```

```
'LCD connection settings
```

```
#define LCD_IO 8
```

```
#define LCD_DATA_PORT PORTC
```

```
#define LCD_RS PORTD.0
```

```
#define LCD_RW PORTD.1
```

```
#define LCD_Enable PORTD.2
```

```
'Main routine
```

```
Do
```

```
Print "Hello World"
```

Wait 1 sec

CLS

Wait 1 sec

Loop

Dim

Dim has two uses, both of which involve large variables. It can be used to define arrays, and to declare variables larger than 1 byte.

Syntax:

For Variables > 1 byte:

Dim *variable* [, *variable2* [, *variable3*]] [As *type*] [Alias *othervar* [, *othervar2*]] [At *location*]

For Arrays:

Dim *array*(*size*) [At *location*]

Explanation:

The Dim variable command is used to inform GCBASIC of variables that are larger than 1 byte, or to create alternate names for other variables.

The Dim array command also sets up array variables. When using it for this, *size* must be a constant value up to 80.

type specifies the type of variable that is to be created. Different variable types can hold values over different ranges, and use different amounts of RAM. See the Variables article for more information.

When multiple variables are included on the one line, GCBASIC will set them all to the type that is specified at the end of the line. If there is no type specified, then GCBASIC will make the variable a byte.

Alias creates a variable using the same memory location as one or more other variables. It is mainly used internally in GCBASIC to treat system variables as a word. For example, this command is used to create a word variable, made up from the two memory locations used to store the result of an A/D conversion:

A variable can be placed at a specific location in the data memory of the chip using the *At* option. *location* will be used whether it is a valid location or not, but a warning will be generated if GCBASIC has already allocated the memory, or if the memory does not appear to be valid. This can be used for peripherals that have multi byte buffers in RAM.

Dim ADResult As Word Alias ADRESH, ADRESL

Example:

'This program will set up an array, and a word variable

```
dim DataList(10)
```

```
dim Reading as word
```

```
Reading = 21978
```

```
DataList(1) = 15
```

Dir

Syntax:

Dir *port.bit* {In | Out} (**Individual Form**)

Dir *port* {In | Out | *DirectionByte*} (**Entire Port Form**)

Command Availability:

Available on all microcontrollers. However, some low end PIC microcontrollers (10F, 12F5x and 16F5x chips) will only accept the entire port form.

Explanation:

The Dir command is used to set the direction of the ports of the microcontroller chip. The individual form sets the direction of one pin at a time, whereas the entire port form will set all bits in a port.

In the individual form, specify the port and bit (ie. PORTB.4), then the direction, which is either In or Out.

The entire port form is similar to the TRIS instruction offered by some PIC chips. To use it, give the name of the port (ie. PORTA), and then a byte is to be written into the TRIS variable. *This form of the command is for those who are familiar with the PIC chip's internal architecture.*

WARNING: PICs use 0 for out and 1 for in. When IN and OUT are used there are no compatibility issues.

Example:

'This program sets PORTA bits 0 and 1 to in, and the rest to out.

'It also sets all of PORTB to output, except for B1.

'Individual form is used for PORTA:

DIR PORTA.0 IN

DIR PORTA.1 IN

DIR PORTA.2 OUT

DIR PORTA.3 OUT

DIR PORTA.4 OUT

DIR PORTA.5 OUT

DIR PORTA.6 OUT

DIR PORTA.7 OUT

'Entire port form used for B:

DIR PORTB b'00000010'

'Entire port form used for C:

DIR PORTC IN

DisplayChar

Syntax:

DisplayChar (*display, character*)

Explanation:

This command will display the given ASCII character on a seven segment LED display. *display* is the number of the display to use, and *character* is the ASCII character to show.

Example:

'This program will show "Hello" on a LED display

'The display should be connected to PORTB

```
#chip 16F877A, 20
```

```
#define DisplayPortA PORTB
```

```
Dim Message As String
```

```
Message = "Hello "
```

```
For Counter = 1 to 6
```

```
    DisplayChar 1, Message(Counter)
```

```
    Wait 250 ms
```

```
Next
```

DisplayValue

Syntax:

DisplayValue *display*, *data*

Explanation:

This command will display the given value on a seven segment LED display. *display* is the number of the display to use, and *data* is the value between 0 and 9 to show

Example:

```
'This program will count from 0 to 99 on two LED displays
```

```
#chip 16F819, 8
```

```
#config osc = int
```

```
#define DISP_SEG_A PORTB.0
```

```
#define DISP_SEG_B PORTB.1
```

```
#define DISP_SEG_C PORTB.2
```

```
#define DISP_SEG_D PORTB.3
```

```
#define DISP_SEG_E PORTB.4
```

```
#define DISP_SEG_F PORTB.5
```

```
#define DISP_SEG_G PORTB.6
```

```
#define DISP_SEL_1 PORTA.0
```

```
#define DISP_SEL_2 PORTA.1
```

```
Do
```

```
For Counter = 0 To 99
```

```
'Get the 2 digits
```

```
Number = Counter
```

```
Num1 = 0
```

```
If Number >= 10 Then
```

```
Num1 = Number / 10
```

```
'SysCalcTempX stores remainder after division
```

```
Number = SysCalcTempX
```

```
End If
```

```
Num2 = Number
```

```
'Show the digits
```

```
'Each DisplayValue will erase the other (multiplexing)
```

```
'So they must be called often enough that the flickering
```

```
'cannot be seen.
```

```
Repeat 500
```

```
DisplayValue 1, Num1
```

```
Wait 1 ms
```

```
DisplayValue 2, Num2
```

```
Wait 1 ms
```

```
End Repeat
```

```
Next
```

```
Loop
```

Do

Syntax:

Do [{While | Until} *condition*]

...

program code

...

Loop [{While | Until} *condition*]

Explanation:

The Do command will cause the code between the Do and the Loop to run repeatedly while *condition* is true or until *condition* is true, depending on whether While or Until has been specified.

Note that the While or Until and the condition can only be specified once, or not at all. If they are not specified, then the code will repeat endlessly.

Example:

'This code will flash a light until the button is pressed

```
#chip 12F629, 4
```

```
#config osc = int
```

```
#define BUTTON GPIO.3
```

```
#define LIGHT GPIO.5
```

```
Dir BUTTON In
```

```
Dir LIGHT Out
```

```
Do Until BUTTON = 1
```

```
PulseOut LIGHT, 1 s
```

```
Wait 1 s
```

```
Loop
```

End

Syntax:

End

Explanation:

When the End command is used, the program will immediately stop running. There are very few cases where this command is needed - generally, the program should be an endless loop.

Example:

'This program will turn on the red light, but not the green light

```
Set RED On
```

```
End
```

```
Set GREEN On
```

EPRead

Syntax:

EPRead *location, store*

Command Availability:

Available on all PIC chips with EEPROM data memory.

Explanation:

EPRead is used to read information from the EEPROM data storage that many microcontroller chips are equipped with. *location* represents the location to read data from, and varies from one chip to another. *store* is the variable in which to store the data after it has been read from EEPROM.

Example:

'Program to turn a light on and off

'Will remember the last status

#chip tiny2313, 1

#define Button PORTB.0

#define Light PORTB.1

Dir Button In

Dir Light Out

'Load saved status

EPRead 0, LightStatus

If LightStatus = 0 Then

Set Light Off

```
Else
Set Light On
End If
```

```
Do
'Wait for the button to be pressed
Wait While Button = On
Wait While Button = Off
```

```
'Toggle value, record
LightStatus = !LightStatus
EPWrite 0, LightStatus
```

```
'Update light
If LightStatus = 0 Then
Set Light Off
Else
Set Light On
End If
Loop
```

EPWrite

Syntax:

EPWrite *location, data*

Command Availability:

Available on all PIC chips with EEPROM data memory.

Explanation:

EPWrite is used to write information to the EEPROM data storage, so that it can be accessed later by a programmer on the PC, or by the EPRead command. *location* represents the location to read data from, and varies from one chip to another. *data* is the data that is to be written to the EEPROM, and can be a value or a variable.

Example:

```
#chip 16F819, 8

#config osc = int

'Set the input pin direction
Dir PORTA.0 In

'Loop to take readings until the EEPROM is full
For CurrentAddress = 0 to 255
  'Take a reading and log it
  EPWrite CurrentAddress, ReadAD(AN0)

  'Wait 10 minutes before getting another reading
  Wait 10 min

Next
```

Exit Sub

Syntax:

Exit Sub

Explanation:

This command will make the program exit the subroutine it is currently in, as it would if it came to the end of the subroutine.

Example:

```
#chip tiny13, 1

#define SENSOR PORTB.0
#define BUZZER PORTB.1
#define LIGHT PORTB.2

Dir SENSOR In
Dir BUZZER Out
Dir LIGHT Out

Do

  Burglar

Loop

'Burglar Alarm subroutine
```

```

Sub Burglar
If SENSOR = 0 Then
Set BUZZER Off
Set LIGHT Off
Exit Sub
End If
Set BUZZER On
Set LIGHT On
End Sub

```

FilledBox

Syntax:

FilledBox(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a filled box on a graphic LCD from the upper corner of X1, Y1 location to X2,Y2 location.

See also Box command

For

Syntax:

For *counter* = *start* To *end* [Step *increment*]

```

...
program code
...

```

Next

Explanation:

The For command is ideal for situations where a piece of code needs to be run a set number of times, and where it is necessary to keep track of how many times the code has run. When the For command is first executed, *counter* is set to *start*. Then, each successive time the program loops, *increment* is added to *counter*, until *counter* is equal to *end*. Then, the program continues beyond the Next.

Step and *increment* are optional. If Step is not specified, GCBASIC will increment *counter* by 1 each time the code is run.

Example:

'This code will flash a green light 6 times.

```
#chip 16F88, 8
```

```
#config Osc = Int
```

```
#define LED PORTB.0
```

```
Dir LED Out
```

```
For LoopCounter = 1 to 6
```

```
PulseOut Led, 1 s
```

```
Wait 1 s
```

```
Next
```

Get

Syntax:

var = Get(Line, Column)

Command Availability:

Available on all microcontrollers, except if the LCD is connected using 2 bit mode.

Explanation:

The Get function reads the ASCII character code at a given location on the LCD.

GLCDCLS

Syntax:

GLCDCLS

Explanation:

Clears the screen of a Graphic LCD.

GLCDPrint

Syntax:

GLCDPrint(PrintLocX, PrintLocY, PrintData)

PrintLocX is the X corrdinate location for the data

PrintLocY is the Y coordinate location for the data

PrintData is a String or String variable of the data to display

Explanation:

Prints string character(s) at a specified location on the GLCD screen.

On a 128x64 GLCD display

X is typically 0 to 128

Y is typically 0 to 64

GLCDDrawChar

Syntax:

GLCDDrawChar(CharLocX, CharLocY, CharCode)

CharLocX is the X corrdinate location for the character

CharLocY is the Y coordinate location for the character

CharCode is the ASCII number of the character to display. Can be decimal hex or binary.

Explanation:

Displays an ASCII character at a specified X and Y location.

On a 128x64 Graphic LCD

X = 1 to 128

Y = 1 to 64

GLCDWriteByte

Syntax:

GLCDWriteByte (LCDByte)

Explanation:

Writes a byte of data to the Graphic LCD memory

GLCDReadByte

Syntax:

GLCDReadByte

Explanation:

Reads a byte of data from the Graphic LCD memory

Gosub

Syntax:

Gosub *label*

Explanation:

The Gosub command is used to jump to a label as a subroutine, in a similar way to Goto. The difference is that Return can then be used to return to the line of code after the Goto.

Note: Gosub should not be used if it can be avoided. It is not required to call a subroutine that has been defined using Sub, just write the name of the subroutine.

Example:

'This program will flash an LED on portb bit 0 and play a beep on
'porta bit 4. until the robot is turned off.

#chip 16F628A, 4 'Change this to suit your circuit

#define SOUNDOUT PORTA.4

#define LIGHT PORTB.0

Dir LIGHT Out

Do

'Flash Light

PulseOut LIGHT, 1's

Wait 1 s

'Beep

Gosub PlayBeep

Loop

PlayBeep:

Tone 200, 10

Tone 100, 10

Return

Goto

Syntax:

`Goto label`

Explanation:

The Goto command will make the robot jump to the line specified, and continue running the program from there. The Goto command is mainly useful for exiting out of loops - if you need to create an infinite loop, use the Do command instead.

Be careful how you use Goto. If used too much, it can make programs very hard to read.

To define a label, put the name of the label alone on a line, with just a colon (:) after it.

Example:

```
'This program will flash the light until the button is pressed
```

```
'off. Notice the label named SWITCH_OFF.
```

```
#chip 16F628A, 4 'Change this line to suit your circuit
```

```
#define BUTTON PORTB.0
```

```
#define LIGHT PORTB.1
```

```
Dir BUTTON In
```

```
Dir BUTTON Out
```

```
Do

PulseOut LIGHT, 500 ms

If BUTTON = 1 Then Goto SWITCH_OFF

Wait 500 ms

If BUTTON = 1 Then Goto SWITCH_OFF

Loop

SWITCH_OFF:

Set LIGHT Off

'Chip will enter low power mode when program ends
```

HPWM

Syntax:

HPWM *channel, frequency, duty cycle*

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command sets up the hardware PWM module of the PIC chip to generate a PWM waveform of the given frequency and duty cycle. Once this command is called, the PWM will be emitted until PWMOFF is called. If you only need one particular frequency and duty cycle, you should use PWMOn and the constants PWM_Freq and PWM_Duty instead.

channel is 1 or 2, and corresponds to the pins CCP1 and CCP2 respectively. On chips with only one CCP port, pin CCP or CCP1 is always used, and *channel* is ignored. (It should be set to 1 anyway to allow for future upgrades to more powerful PIC chips.)

frequency sets the frequency of the PWM output. It is measured in KHz. The maximum value allowed is 255 KHz. The minimum value varies depending on the clock speed. 1 KHz is the minimum on chips 16 MHz or under and 2 KHz is the lowest possible on 20 MHz chips. In situations that do not require a specific PWM frequency, the PWM frequency should equal approximately 1 five-hundredth the clock speed of the PIC (ie 40 KHz on a 20 MHz chip, 16 KHz on an 8 MHz chip). This gives the best duty cycle resolution possible.

duty cycle specifies the desired duty cycle of the PWM signal, and ranges from 0 to 255 where 255 is 100% duty cycle.

Example:

```
'This program will alter the brightness of an LED using
'hardware PWM.

'Select chip model and speed
#chip 16F877A, 20

'Set the CCP1 pin to output mode
DIR PORTC.2 out

'Main code
```

```
do
    'Turn up brightness over 2.5 seconds
    For Bright = 1 to 255
        HPWM 1, 40, Bright
    wait 10 ms
    next
    'Turn down brightness over 2.5 seconds
    For Bright = 255 to 1
        HPWM 1, 40, Bright
    wait 10 ms
    next
loop
```

HSerPrint

HSerPrint is used to send a value over the serial connection. *value* can be a string, word or byte - SerPrint is very similar to Print. *channel* is the serial connection to send data through.

Syntax:

HSerPrint *value*

Command Availability:

Available on all microcontrollers with a USART or UART module.

Note:

HSerPrint will not send any new line characters. If the chip is sending to a terminal, these commands should follow every SerPrint:

HSerPrint 13
HSerPrint 10

Or

HSerPrintCRLF

Example:

'This program will display any values received over the serial
'connection. If "pot" is received, the value of the analog sensor
'will be sent.

'Note: This has been adapted from the SerPrint example.

'Chip settings

#chip 18F2525, 8

#config Osc = Int

'LCD settings

#define LCD_IO 4

#define LCD_RS PORTC.7

#define LCD_RW PORTC.6

#define LCD_Enable PORTC.5

#define LCD_DB4 PORTC.4

#define LCD_DB5 PORTC.3

#define LCD_DB6 PORTC.2

#define LCD_DB7 PORTC.1

'USART settings

```

#define USART_BAUD_RATE 9600

'Potentiometer

#define POT_PORT PORTA.0
#define POT_AN AN0

'Set pin directions
Dir SerOutPort Out
Dir SerInPort In
Dir POT_PORT In

'Create buffer variables to store received messages
Dim Buffer As String
Dim OldBuffer As String
BufferSize = 0

'Show test messages
Print "Serial Tester"
Wait 1 s
HSerPrint "GCBASIC RS232 Test"
HSerSend 13
HSerSend 10
Wait 1 s

'Main loop

```

```

Do

'Get a byte from the terminal

HSerReceive Temp

'If Enter key was pressed, deal with buffer contents

If Temp = 13 Then

Buffer(0) = BufferSize

'Try to execute commands in buffer

If Not ExecCommand (Buffer) Then

'Show message on bottom line, last message on top.

CLS

Print OldBuffer

Locate 1, 0

Print Buffer

'Store the message for next time

OldBuffer = Buffer

End If

BufferSize = 0

End If

'Backspace code, delete last character in buffer

If Temp = 8 Then

If BufferSize > 0 Then BufferSize -= 1

End If

```

```

'Received ASCII code between 32 and 127, add to buffer

If Temp >= 32 And Temp <= 127 Then

BufferSize += 1

Buffer(BufferSize) = Temp

End If

Loop

'Takes a sensor reading and sends it to terminal

Sub SendSensorReading

HSerPrint "Sensor Reading: "

HSerPrint ReadAD10(POT_AN)

HSerSend 13

HSerSend 10

End Sub

'Will check the buffer for a command

'If command found, run it and return true

'If not, return false

Function ExecCommand (CmdIn As String)

ExecCommand = False

'If received command is "pot", show potentiometer value

If CmdIn = "pot" Then

SendSensorReading

ExecCommand = True

End If

End Function

```

HSerReceive

Syntax:

Used as subroutine:

HSerReceive output

Used as function:

output = HSerReceive

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will read a byte from the hardware RS232 module. It can be used either as a subroutine or as a function. If used as a subroutine, a variable must be supplied to store the received value in. If used as a function, it will return the received value.

Example:

```
'This program will read a value from the USART, and display it on  
PORTB.
```

```
#chip 16F877A, 20
```

```
'USART settings
```

```
#define USART_BAUD_RATE 9600
```

```
'Set PORTB to input
```

```
Dir PORTB Out
```

```
'Set USART receive pin to input
```

```
Dir PORTC.7 In
```

```
'Main loop
```

```
Do
```

```
'Get and display value
```

```
'If there is no new data, HSerReceive will return old value.
```

```
HSerReceive PORTB
```

```
'Could also write:
```

```
'PORTB = HSerReceive
```

```
Loop
```

HSerSend

Syntax:

HSerSend *data*

Command Availability:

Available on all microcontrollers with a USART or UART module.

Explanation:

This command will send a byte given by *data* using the hardware RS232 module.

Example:

```
'This program will send the status of PORTB through the hardware  
'serial module.
```

```
#chip 16F877A, 20
```

```
'USART settings
```

```
#define USART_BAUD_RATE 9600
```

```
'Set PORTB to input
```

```
Dir PORTB In
```

```
'Set USART transmit pin to output
```

```
Dir PORTC.6 Out
```

```
'Main loop
```

```
Do
```

```
'Send PORTB value through USART
```

```
HSerSend PORTB
```

```
'Short delay for receiver to process message
```

```
Wait 10 ms
```

```
Loop
```

I2CReceive

Syntax:

I2CReceive *data*

I2CReceive *data*, *ack*

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

The I2CReceive command will send *data* through the I2C connection. If *ack* is TRUE, or no value is given for *ack*, then I2CReceive will send an ack.

If in master mode, I2CReceive will read the data immediately. If in slave mode, I2CReceive will wait for the master to send the data before reading.

Example:

```
'This program reads an I2C register and sets an LED if it is over 100.
```

```
'It will read from address 83, register 1.
```

```
'Chip settings
```

```
#chip 18F2620, 8
```

```
'I2C settings
```

```

#define I2C_MODE Master

#define I2C_DATA PORTC.0

#define I2C_CLOCK PORTC.1


'Misc settings

#define LED PORTB.0


'Main loop
Do
'Send start
I2CStart

'Request value
I2CSend 83
I2CSend 1

'Read value
I2CReceive ValueIn

'Send stop
I2CStop


'Turn on LED if received value > 100
Set LED Off

If ValueIn > 100 Then Set LED On

```

```
'Delay
```

```
Wait 20 ms
```

```
Loop
```

I2CSend

Syntax:

```
I2CSend data
```

```
I2CSend data, ack
```

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

The I2CSend command will send *data* through the I2C connection. If *ack* is TRUE, or no value is given for *ack*, then I2CSend will wait for an Acknowledge bit from the receiver before continuing.

If in master mode, I2CSend will send the data immediately. If in slave mode, I2CSend will wait for the master to request the data before sending.

Example:

```
'This program will act as an I2C analog to digital converter
```

```
'When data is requested from address 83, registers 0 through
```

'3, it will return the value of AN0 through AN3.

'Chip model

#chip 16F88, 8

'I2C settings

#define I2C_MODE Slave

#define I2C_CLOCK PORTB.0

#define I2C_DATA PORTB.1

'Main loop

Do

'Wait for start condition

I2CStart

'Get address

I2CReceive Address

If Address = 83 Then

'If address was this device's address, respond

I2CReceive Register

OutValue = ReadAD(Register)

I2CSend OutValue

End If

I2CStop

Wait 5 ms

Loop

I2CStart

Syntax:

I2CStart

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

If the I2C routines are operating in Master mode, this command will send a start condition. If routines are in Slave mode, it will pause the program until a start condition is sent by the master. It should be placed at the start of every I2C transmission.

If interrupt handling is enabled, this command will disable it.

Example:

Please see I2CSend and I2CReceive for an example.

I2CStop

Syntax:

I2CStop

Command Availability:

Available on all microcontrollers except 12 bit instruction PICs (10F, 12F5xx, 16F5xx chips)

Explanation:

When in Master mode, this command will send an I2C stop condition, and re-enable interrupts if I2CStart disabled them. In Slave mode, it will re-enable interrupts.

I2CStop should be called at the end of every I2C transmission.

Example:

Please see I2CSend and I2CReceive for an example.

If Then Else

Syntax:

Short form:

If *condition* Then *command*

Long form:

If *condition* Then

...

program code

...

End If

Using Else:

If *condition* Then

code to run if true

Else

code to run if false

End If

Explanation:

The If command is the most common command used to make decisions. If *condition* is true, then *command* (short) or *program code* (long) will be run. If it is false, then the robot will skip to the code located on the next line (short) or after the End If (long form).

If Else is used, then the condition between If and Else will run if the condition is true, and the code between Else and End If will run if the condition is false.

Example:

'Turn a light on or off depending on a light sensor

```
#chip 12F683, 8
```

```

#config osc = int

#define LIGHT GPIO.1

#define SENSOR AN3

#define SENSOR_PORT GPIO.4

Dir LIGHT Out

Dir SENSOR_PORT In

Do

If ReadAD(SENSOR) > 128 Then

Set LIGHT Off

Else

Set LIGHT On

End If

Loop

```

InitGLCD

Syntax:

InitGLCD

Explanation:

This initializes the Graphical LCD for operation. Here are the steps it takes:

'Set pin directions

Dir GLCD_RS Out
Dir GLCD_RW Out
Dir GLCD_ENABLE Out
Dir GLCD_CS1 Out
Dir GLCD_CS2 Out
Dir GLCD_RESET Out

'Reset

Set GLCD_RESET Off
Wait 1 ms
Set GLCD_RESET On
Wait 1 ms

'Select both chips

Set GLCD_CS1 On
Set GLCD_CS2 On

'Set on

Set GLCD_RS Off
GLCDWriteByte 63

'Set Z to 0

GLCDWriteByte 192

'Deselect chips

Set GLCD_CS1 Off
Set GLCD_CS2 Off

'Clear screen

GLCDCLS

InitSer

Syntax:

InitSer *channel, rate, start, data, stop, parity, invert*

Explanation:

This command will set up the serial communications. The parameters are as follows:

channel is 1, 2 or 3, and refers to the I/O ports that are used for communication.

rate is the bit rate, which is given by the letter r and then the desired rate in bps. Acceptable units are r300, r600, r1200, r2400, r4800, r9600 and r19200.

start gives the number of start bits, which is usually 1. To make the PIC wait for the start bit before proceeding with the receive, add 128 to *start*. (Note: it may be desirable to use the WaitForStart constant here.)

data tells the program how many data bits are to be sent or received. In most situations this is 8, but it can range between 1 and 8, inclusive.

stop is the number of stop bits. If *start* bit 7 is on, then this number will be ignored.

parity refers to a system of error checking used by many devices. It can be odd (in which there must always be an odd number of high bits), even (where the number of high bits must always be even), or none (for systems that do not use parity).

invert can be either "normal" or "invert". If it is "invert", then high bits will be changed to low, and low to high.

Example:

Please refer to SerSend for an example of InitSer

InitTimer0

Syntax:

InitTimer0 *source*, *prescaler*

Command Availability:

Available on all microcontrollers with a Timer 0 module.

Explanation:

InitTimer0 will set up timer 0, according to the settings given. *source* can be Osc or Ext.

On a PIC microcontroller, *prescaler* can be one of the following settings:

- PS0_2
- PS0_4
- PS0_8
- PS0_16
- PS0_32
- PS0_64
- PS0_128
- PS0_256

These correspond to a prescaler of between 1/2 and 1/256 the oscillator speed. The prescaler will apply to either the oscillator or the external clock input.

When the timer overflows from 255 to 0, a Timer0Overflow interrupt will be generated. This can be used in conjunction with On Interrupt to run a section of code periodically.

Example:

'This code will use Timer 0 and On Interrupt to generate a Pulse Width Modulation signal, that will allow the speed of a motor to be easily controlled.

```
#chip 16F88, 8
```

```
#config osc = int
```

```
#define MOTOR PORTB.0
```

'Call the initialisation routine

```
InitMotorControl
```

'Main routine

Do

'Increase speed to full over 2.5 seconds

For Speed = 0 to 100

```
MotorSpeed = Speed
```

```
Wait 25 ms
```

Next

'Hold speed

```
Wait 1 s
```

```

'Decrease speed to zero over 2.5 seconds

For Speed = 100 to 0

MotorSpeed = Speed

Wait 25 ms

Next

'Hold speed

Wait 1 s

Loop

'Setup routine

Sub InitMotorControl

'Clear variables

MotorSpeed = 0

PWMCOUNTER = 0

'Add a handler for the interrupt

On Interrupt Timer0Overflow Call PWMHandler

'Set up the timer

InitTimer0 Osc, PS0_1/2

'Timer 0 starts automatically on a PIC

End Sub

'PWM sub

'This will be called when Timer 0 overflows

```

```

Sub PWMHandler

If MotorSpeed > PWMCounter Then

Set MOTOR On

Else

Set MOTOR Off

End If

PWMCounter += 1

If PWMCounter = 100 Then PWMCounter = 0

End Sub

```

InitTimer1

Syntax:

InitTimer1 *source*, *prescaler*

Command Availability:

Available on all microcontrollers with a Timer 1 module.

Explanation:

InitTimer1 will set up timer 1, according to the settings given. *source* can be Osc , Ext or ExtOsc (ExtOsc is only available on PIC microcontrollers). On a PIC, *prescaler* can be one of the following settings:

- PS1_1
- PS1_2

- PS1_4

- PS1_8

Example:

'This example will measure that time that a switch stays on for

#chip 16F819, 20

#define Switch PORTA.0

Dir Switch In

DataCount = 0

InitTimer Osc, PS1_8

Dim TimerValue As Word

Do

ClearTimer 1

Wait Until Switch = On

StartTimer 1

Wait Until Switch = Off

StopTimer 1

'Read the timer

TimerValue = Timer1

'Log the timer value

```
EPWrite(DataCount, TimerValue_H)

EPWrite(DataCount + 1, TimerValue)

DataCount += 2

Loop
```

INKEY

Syntax:

output = INKEY

Explanation:

The INKEY function will read the last pressed key from a PS/2 keyboard, and return an ASCII value corresponding to the key. If no key is pressed, then INKEY will return 0.

It will also monitor Caps Lock, Num Lock and Scroll Lock keys, and update the status LEDs as appropriate.

Example:

'A program to accept messages from a standard PS/2 keyboard

'Any keys pressed will be shown on an LCD screen.

'Hardware settings

#chip 18F4620, 20

'LCD connection settings

```

#define LCD_IO 4

#define LCD_DB4 PORTD.4

#define LCD_DB5 PORTD.5

#define LCD_DB6 PORTD.6

#define LCD_DB7 PORTD.7

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2


'PS/2 connection settings

#define PS2Clock PORTC.1

#define PS2Data PORTC.0

#define PS2_DELAY 10 ms


'Set up key log

Dim KeyLog(32)

DataCount = 0

KeyLog(1) = 32


Main:

'Read the last pressed key

KeyIn = INKEY

'If no key pressed, try reading again

If KeyIn = 0 Then Goto Main

```

```

'Escape pressed - clear message

If KeyIn = 27 Then

DataCount = 0

For DataPos = 1 to 32

KeyLog(DataPos) = 32

Next

Goto DisplayData

End If


'Backspace pressed - delete last character

If KeyIn = 8 Then

If DataCount = 0 Then Goto Main

KeyLog(DataCount) = 32

DataCount = DataCount - 1

Goto DisplayData

End If


'Otherwise, add the character to the buffer

If KeyIn >= 31 And KeyIn <= 127 Then

DataCount = DataCount + 1

KeyLog(DataCount) = KeyIn

End If

DisplayData:

'Display key buffer

'LCDWriteChar is used instead of Print for greater control

CLS

```

```
For DataPos = 1 to DataCount  
  
If DataPos = 17 then Locate 1, 0  
  
LCDWriteChar KeyLog(DataPos)  
  
Next  
  
Goto Main
```

Interrupts

About Interrupts

Interrupts are a feature of many microcontrollers. They allow the microcontroller to temporarily pause (interrupt) the code it is running and then start running another piece of code when some event occurs. Once it has dealt with the event, it will return to where it was and continue running the program.

Many events can trigger an interrupt, such as a timer reaching its limit, a serial message being received, or a special pin on the microcontroller receiving a signal.

Using Interrupts

There are two ways to use interrupts in GCBASIC. The first way is to use the On Interrupt command. This will automatically enable a given interrupt, and run a particular subroutine when the interrupt occurs.

The other way to deal with interrupts is to create a subroutine called Interrupt. GCBASIC will call this subroutine whenever an interrupt occurs, and then your code can check the "flag" bits to determine which interrupt has occurred, and what should be done about it. If you use this approach, then you'll need to enable the desired interrupts manually. It is also essential that your code clears the flag bits, or else the interrupt routine will be called repeatedly.

Some combination of these two methods is also possible - the code generated by On Interrupt with check to see if the interrupt is one it recognises. If the interrupt is recognised, On Interrupt will deal with it - if not, the Interrupt subroutine will be called to deal with the interrupt.

The recommended way is to use On Interrupt, as it is both more efficient and easier to set up.

During some sections of code, it is desirable not to have any interrupts occur. If this is the case, then use the IntOff command to disable interrupts at the start of the section, and IntOn to re-enable them at the end. If any interrupt events occur while interrupts are disabled, then they will be processed as soon as interrupts are re-enabled. If the program does not use interrupts, IntOn and IntOff will be removed automatically by GCBASIC.

IntOff

Syntax:

IntOff

Command Availability:

Available on PIC microcontrollers with interrupt support. Will be automatically removed on chips without interrupts.

Explanation:

IntOff is used to disable interrupts on the microcontroller. It should be used at the start of code which is timing-sensitive, and which would not function correctly if paused and restarted.

It is essential that IntOn is used to turn interrupts on again after the timing-sensitive code has finished running. If not, no interrupts will be handled.

IntOff will be removed from the program if no interrupts are used. It is recommended that IntOff be placed before all code that is timing sensitive, in case interrupts are implemented later.

IntOn

Syntax:

IntOn

Command Availability:

Available on PIC microcontrollers with interrupt support. Will be automatically removed on chips without interrupts.

Explanation:

IntOn is used to enable interrupts on the microcontroller after IntOff has disabled them. It should be used at the end of code which is timing-sensitive.

IntOn will be removed from the program if no interrupts are used.

KeypadData

Syntax:

var = KeypadData

Explanation:

This function will return a value corresponding to the key that is pressed on the keypad. Note that if two or more keys are pressed, then only one value will be returned.

var can have one of the following values:

Value	Constant Name	Key Pressed
0		0
1		1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10	KEY_A	A
11	KEY_B	B
12	KEY_C	C
13	KEY_D	D
14	KEY_STAR	Asterisk/Star (*)
15	KEY_HASH	Hash (#)
255	KEY_NONE	None

Example:

'Program to show the value of the last pressed key on the LCD

#chip 18F4550, 20

'LCD connection settings

```

#define LCD_IO 4

#define LCD_DB4 PORTD.4

#define LCD_DB5 PORTD.5

#define LCD_DB6 PORTD.6

#define LCD_DB7 PORTD.7

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2


'Keypad connection settings

#define KeypadPort PORTB


'Main loop

Do

'Get key

Temp = KeypadData


'If a key is pressed, then display it

If Temp <> KEY_NONE Then

CLS

Print Temp

Wait 100 ms

End If

Loop

```

KeypadRaw

Syntax:

largevar = KeypadRaw

Explanation:

This function will return a 16 bit value, in which each bit corresponds to a key on the keypad. If the key is pressed its bit will hold 1, and if it is released its bit will contain a 0.

This table shows the key that each bit corresponds to:

Bit	Key Position (row, col)	Common Key Symbol
15	1,1	1
14	1,2	2
13	1,3	3
12	1,4	A
11	2,1	4
10	2,2	5
9	2,3	6
8	2,4	B
7	3,1	7
6	3,2	8
5	3,3	9
4	3,4	C
3	4,1	*
2	4,2	0
1	4,3	#
0	4,4	D

Example:

'Program to show the keypad status using LEDs

#chip 16F877A, 20

'Keypad connection settings

#define KeypadPort PORTB

'LEDs

#define LED1 PORTC

#define LED2 PORTD

Dir LED1 Out

Dir LED2 Out

'Declare a 16 bit variable for the key value

Dim KeyStatus As Word

'Main loop

Do

'Get key

KeyStatus = KeypadRaw

'Display

LED1 = KeyStatus_H 'High Byte

LED2 = KeyStatus 'Low Byte

Loop

LCDCreateChar

Syntax:

`LCDCreateChar char, chardata()`

Explanation:

The `LCDCreateChar` command is used to send a custom character to the LCD.

Each character on the LCD is made up from an 8 row by 5 column (5x8) matrix of pixels. The data to be sent to the LCD is composed of an 8 element array, where each element corresponds to a row. Inside each element, the 5 lowest bits make up the data for the corresponding row. When a bit is set a dot will be drawn at the matching location; when it is cleared, no dot will appear.

An array of more than 8 elements may be used, but only the first 8 will be read.

char is the ASCII value of the character to create. ASCII codes 0 through 7 are usually used to store custom characters.

chardata() is an array containing the data for the character.

Example:

```
'This program draws a smiling face character
```

```
'General hardware configuration
```

```
#chip 16F877A, 20
```

```
'LCD connection settings
```

```
#define LCD_IO 8
```

```
#define LCD_DATA_PORT PORTC
```

```
#define LCD_RS PORTD.0
```

```
#define LCD_RW PORTD.1
```

```
#define LCD_Enable PORTD.2
```

```
'Create an array to store the character until it is copied
```

```
Dim TempArray(8)
```

```
'Set the array to hold the character
```

```
'Binary has been used to improve the readability of the code, but is  
not essential
```

```
TempArray(1) = b'00011011'
```

```
TempArray(2) = b'00011011'
```

```
TempArray(3) = b'00000000'
```

```
TempArray(4) = b'00000100'
```

```
TempArray(5) = b'00000000'
```

```
TempArray(6) = b'00010001'
```

```
TempArray(7) = b'00010001'
```

```
TempArray(8) = b'00001110'
```

```
'Copy the character from the array to the LCD
```

```
LCDCreateChar 0, TempArray()
```

```
'Draw the custom character
```

```
LCDWriteChar 0
```

LCDHex

Syntax:

LCDHex *value*

Explanation:

The LCDHex command will show the specified value on the LCD, at the current cursor position. It will convert the byte it has been given into hexadecimal format.

Example:

'A program to count from 0 to FF on an LCD screen.

'General hardware configuration

```
#chip 16F877A, 20
```

'LCD connection settings

```
#define LCD_IO 8
```

```
#define LCD_DATA_PORT PORTC
```

```
#define LCD_RS PORTD.0
```

```
#define LCD_RW PORTD.1
```

```
#define LCD_Enable PORTD.2
```

```
For Counter = 0 To 255
```

```
CLS
```

```
LCDHex Counter
```

```
Wait 250 ms
```

```
Next
```

LCDWriteChar

Syntax:

LCDWriteChar *char*

Explanation:

The LCDWriteChar command will show the specified character on the LCD, at the current cursor position.

char is the ASCII value of the character to show. On most LCDs, characters 0 through 7 are user defined, and can be set using the LCDCreateChar command.

Example:

```
'This program draws a smiling face character
```

```
'General hardware configuration
```

```
#chip 16F877A, 20
```

```
'LCD connection settings
```

```
#define LCD_IO 8
```

```
#define LCD_DATA_PORT PORTC
```

```
#define LCD_RS PORTD.0
```

```
#define LCD_RW PORTD.1
```

```
#define LCD_Enable PORTD.2
```

```
'Create an array to store the character until it is copied
```

```
Dim TempArray(8)
```

```
'Set the array to hold the character
```

```
TempArray(1) = b'00011011'
```

```
TempArray(2) = b'00011011'
```

```
TempArray(3) = b'00000000'
```

```
TempArray(4) = b'00000100'
```

```
TempArray(5) = b'00000000'
```

```
TempArray(6) = b'00010001'
```

```
TempArray(7) = b'00010001'
```

```
TempArray(8) = b'00001110'
```

```
'Copy the character from the array to the LCD
```

```
LCDCreateChar 0, TempArray()
```

'Draw the custom character

LCDWriteChar 0

Line

Syntax:

Line(LineX1,LineY1, LineX2, LineY2, Optional LineColour = 1)

Explanation:

Draws a line on a graphic LCD from X1, Y1 location to X2,Y2 location. Must be horizontal or vertical. Diagonal will not work properly.

Locate

Syntax:

Locate *line, column*

Explanation:

The Locate command is used to move the cursor on the LCD to the given location.

Example:

```

'A Hello World program for GCBASIC.

'Uses Locate to show "World" on the second line


'General hardware configuration

#chip 16F877A, 20


'LCD connection settings

#define LCD_IO 8

#define LCD_DATA_PORT PORTC

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2


'Main routine

Print "Hello"

Locate 1, 5

Print "World"

```

On Interrupt

Syntax:

On Interrupt *event* Call *handler*

On Interrupt *event* Ignore

Command Availability:

Available on PIC microcontrollers with interrupt support.

Explanation:

On Interrupt will add code to call the subroutine *handler* whenever the interrupt *event* occurs. When Call is specified, GCBASIC will enable the interrupt, and call the interrupt handler when it occurs. When Ignore is specified, GCBASIC will disable the interrupt handler and prevent it from being called when the event occurs. If the event occurs while the handler is disabled, then the handler will be called as soon as it is re-enabled. The only way to prevent this from happening is to manually clear the flag bit for the interrupt.

There are many possible interrupt events that can occur, and the events vary greatly from chip to chip. GCBASIC will display an error if a given chip cannot support the specified event.

In some cases, On Interrupt will not be able to set or clear the interrupt flag and/or enable bits. If this is the case, GCBASIC will display a warning. You will need to consult the chip datasheet and use the Set command to manually set/clear the flag and enable bits, both at the start of the program and inside the interrupt handler subroutine.

If On Interrupt is used to handle an event, then the Interrupt subroutine will not be called for that event. However, it will still be called for any events not dealt with by On Interrupt.

Events:

GCBASIC supports the events shown on the table below. Some events are only implemented on a few specialized chips. Events in **bold** are only supported by some PICs.

Note that GCBASIC doesn't fully support all of the hardware which can generate interrupts - some work may be required with various system variables to control the unsupported peripherals.

Event Name	Description
ADCReady	The analog/digital converter has finished a conversion
CANActivity	CAN bus activity is taking place
CANBadMessage	A bad CAN message has been received
CANError	Some CAN error has occurred
CANHighWatermark	CAN high watermark reached
CANRx0Ready	New message present in buffer 0
CANRx1Ready	New message present in buffer 1
CANRx2Ready	New message present in buffer 2
CANRxReady	New message present
CANTx0Ready	Buffer 0 has been sent
CANTx1Ready	Buffer 1 has been sent
CANTx2Ready	Buffer 2 has been sent
CANTxReady	Sending has completed
CCP1	The CCP1 module has captured an event
CCP2	The CCP2 module has captured an event
CCP3	The CCP3 module has captured an event
CCP4	The CCP4 module has captured an event
CCP5	The CCP5 module has captured an event
Comp0Change	The output of comparator 0 has changed
Comp1Change	The output of comparator 1 has changed
Comp2Change	The output of comparator 2 has changed
Crypto	The KEELOQ module has generated an interrupt
EEPROMReady	An EEPROM write has finished
Ethernet	The Ethernet module has generated an interrupt. This must be dealt with in the handler.
ExtInt0	External Interrupt pin 0 has been ed
ExtInt1	External Interrupt pin 1 has been ed
ExtInt2	External Interrupt pin 2 has been ed
ExtInt3	External Interrupt pin 3 has been ed
GPIOChange	The pins on port GPIO have changed
LCDReady	The LCD is about to draw a segment
LPWU	The Low Power Wake Up has been ed

OscillatorFail	The external oscillator has failed, and the PIC is running from an internal oscillator.
PMPReady	A Parallel Master Port read or write has finished
PORTAChange	The pins on port A have changed
PORTABChange	The pins on port A and/or B have changed
PORTBChange	The pins on port B have changed
PSPReady	A Parallel Slave Port read or write has finished
PWMTimeBase	The PWM time base matches the PWM Time Base Period register (PTPER)
SPPReady	A SPP read or write has finished
SSP1Collision	SSP1 has detected a bus collision
SSP1Ready	The SSP/SSP1/MSSP1 module has finished sending or receiving
SSP2Collision	SSP2 has detected a bus collision
SSP2Ready	The SSP2/MSSP2 module has finished sending or receiving
Timer0Overflow	Timer 0 has overflowed
Timer1Overflow	Timer 1 has overflowed
Timer2Overflow	Timer 2 has overflowed
Timer3Overflow	Timer 3 has overflowed
Timer4Overflow	Timer 4 has overflowed
Timer5CAP1	An input on the CAP1 pin has caused the value of Timer 5 to be captured in CAP1BUF
Timer5CAP2	An input on the CAP2 pin has caused the value of Timer 5 to be captured in CAP2BUF
Timer5CAP3	An input on the CAP3 pin has caused the value of Timer 5 to be captured in CAP3BUF
Timer5Match	Timer5 matches the PR5 register
UsartRX1Ready	UART/USART 1 has received data
UsartRX2Ready	UART/USART 2 has received data
UsartTX1Ready	UART/USART 1 is ready to send data
UsartTX2Ready	UART/USART 2 is ready to send data
USB	The USB module has generated an interrupt. This must be dealt with in the handler.
VoltageFail	The input voltage has dropped too low

Example:

'This program increments a counter every time Timer1 overflows

#chip 16F877A, 20

'LCD connection settings

#define LCD_IO 4

#define LCD_DB4 PORTD.4

#define LCD_DB5 PORTD.5

#define LCD_DB6 PORTD.6

#define LCD_DB7 PORTD.7

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2

InitTimer1 Osc, PS1_1/8

StartTimer 1

CounterValue = 0

Wait 100 ms

Print "Int Test"

On Interrupt Timer1Overflow Call IncCounter

Do

CLS

```
Print CounterValue
```

```
Wait 100 ms
```

```
Loop
```

```
Sub IncCounter
```

```
CounterValue ++
```

```
End Sub
```

For more help, see:

The [InitTimer0](#) article contains an example of using Timer 0 and On Interrupt to generate a Pulse Width Modulation signal to control a motor.

Peek

Syntax:

OutputVariable = Peek (*location*)

Explanation:

The Peek function is used to read information from the on-chip RAM of the microcontroller.

location is a word variable that gives the address to read. The exact range of valid values varies from chip to chip.

This command should not normally be used, as it will make the porting of code to another chip very difficult.

Example:

"This program will read and check a value from PORTA

"Will only work on some PICs

Temp = PEEK(5)

IF Temp.2 ON THEN SET green ON

IF Temp.2 OFF THEN SET red ON

Poke

Syntax:

Poke (*location*, *value*)

Explanation:

The Poke command is used to write information to the on-chip RAM of the microcontroller.

location is a word variable that gives the address to write. The exact range of valid values varies from chip to chip.

value is the data to write to the location.

This command should not normally be used, as it will make the porting of code to another chip very difficult.

Example:

This program will set all of the PORTB pins high

```
POKE (6, 255)
```

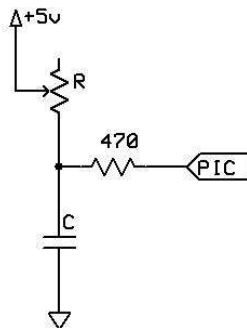
Pot

Syntax:

Pot *pin*, *output*

Explanation:

Pot makes it possible to measure an analog resistance with a digital port, with the addition of a small capacitor. This is the required circuit:



The command works by using the microcontroller pin to discharge the capacitor, then measuring the time taken for the capacitor to charge again through the resistor.

The value for the capacitor must be adjusted depending on the size of the variable resistor. The charging time needs to be approximately 2.5 ms when the resistor is at its maximum value. For a typical 50 k potentiometer or LDR, a 50 nf capacitor is required.

This command should be used carefully. Each time it is inserted, 20 words of program memory are used on the chip, which as a rough guide is more than 15 times the size of the Set command.

pin is the port connected to the circuit. The direction of the pin will be dealt with by the Pot command.

output is the name of the variable that will receive the value.

Example:

```
'This program will beep whenever a shadow is detected
'A potentiometer is used to adjust the threshold

#chip 16F628A, 4
#config INTOSC_OSC_NOCLKOUT
#define ADJUST PORTB.0
#define LDR PORTB.1
#define SoundOut PORTB.2

Dir SoundOut Out

Do

Pot ADJUST, Threshold
Pot LDR, LightLevel

If LightLevel > Threshold Then

Tone 1000, 100

End If

Loop
```

Print

Syntax:

Print *string*

Print *byte*

Print *word*

Print *integer*

Explanation:

The Print command will show the contents of a variable on the LCD. It can display string, word or byte variables.

Example:

```
'A Light Meter program.

'General hardware configuration
#chip 16F877A, 20
#define LightSensor AN0

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
```

```
#define LCD_Enable PORTD.2
```

```
CLS
```

```
Print "Light Meter:
```

```
Locate 1,2
```

```
Print "A GCBASIC Demo"
```

```
Wait 2 s
```

```
Do
```

```
CLS
```

```
Print "Light Level: "
```

```
Print ReadAD(LightSensor)
```

```
Wait 250 ms
```

```
Loop
```

ProgramErase

Syntax:

ProgramErase(*location*)

Command Availability:

Available on all PIC chips with self write capability.

Explanation:

ProgramErase erases information from the program memory on chips that support this feature. The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet.

This command must be called before writing to a block of memory. It is slow in comparison to other GCBASIC commands. Note that it erases memory in 32-byte blocks - see the relevant PIC datasheet for more information.

This is an advanced command which should only be used by advanced developers. Care must be taken with this command, as it can easily erase the program that is running on the PIC.

ProgramRead

Syntax:

ProgramRead (*location*, *store*)

Command Availability:

Available on all PIC chips with self write capability.

Explanation:

ProgramRead reads information from the program memory on chips that support this feature. *location* and *store* are both word variables, meaning that they can store values over 255.

The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet. *store* is 14 bits wide, and thus can store values up to 16383.

This is an advanced command which should only be used by advanced developers.

ProgramWrite

Syntax:

ProgramWrite (*location*, *value*)

Command Availability:

Available on all PIC chips with self write capability.

Explanation:

ProgramWrite writes information to the program memory on chips that support this feature. *location* and *value* are both word variables.

The largest value possible for *location* depends on the amount of program memory on the PIC, which is given on the datasheet. *value* is 14 bits wide, and thus can store values up to 16383.

This is an advanced command which should only be used by advanced developers. ProgramErase must be used to clear a block of memory BEFORE ProgramWrite is called.

PS2ReadByte

Syntax:

output = PS2ReadByte

Explanation:

PS2ReadByte will read a byte from the PS/2 bus. It will return the byte, or 0 if no data was returned by the PS/2 device.

The PS/2 bus will normally be held in the inhibit state. PS2ReadByte will uninhibit the bus for 25 ms. If a response is received, it will be read. Then, the bus will be placed back in the inhibit state.

Example:

For an example, please refer to the INKEY function in the ps2.h file.

PS2SetKBleds

Syntax:

PS2SetKBleds (*LedStatus*)

Explanation:

This routine will turn the status LEDs on a keyboard on or off. *LedStatus* is a variable, of which the lower 3 bits correspond to the 3 LEDs. Bit 0 is for Scroll Lock, bit 1 controls Num Lock and bit 2 controls Caps Lock.

Note that this routine does not alter the status variables within the INKEY routine - so even if the Caps Lock LED is turned on, Caps Lock will stay off.

Example:

```
'A spinning LED program for a keyboard  
  
'Will flash Num Lock, then Caps Lock, then Scroll Lock.  
  
'Hardware settings  
  
#chip 16F88, 8
```

```

#define PS2Clock PORTB.2

#define PS2Data PORTB.3

#define PS2_DELAY 10 ms


'Main Loop

Do

'Turn on only Num Lock (bit 1)

PS2SetKBLeds b'00000010'

Wait 250 ms

'Turn on only Caps Lock (bit 2)

PS2SetKBLeds b'00000100'

Wait 250 ms

'Turn on only Scroll Lock (bit 0)

PS2SetKBLeds b'00000001'

Wait 250 ms

Loop

```

PS2WriteByte

Syntax:

PS2WriteByte *data*

Explanation:

PS2WriteByte will send a byte to a PS/2 device. Once the byte has been written, the PS/2 bus will be placed in the inhibit state.

Example:

For an example, please refer to the PS2SetKBLeds function in the ps2.h file.

PSet

Syntax:

PSet(GLCDX, GLCDY, GLCDState)

Explanation:

Sets or Clears a Pixel at the specified X, Y location. A one for GLCDState sets the pixel and a zero clears the pixel.

PulseOut

Syntax:

PulseOut *pin, time units*

Explanation:

The PulseOut command will set the specified pin high, wait for the specified amount of time, and then set the pin low again. The pin is specified in the same way as it is for the Set command, and the time is the same as for the Wait command.

Example:

'This program flashes an LED on GPIO.0 using PulseOut

#chip 12F629, 4

#config INTRC_OSC_NOCLKOUT

Dir GPIO.0 Out

Do

PulseOut GPIO.0, 1 sec 'Turn LED on for 1 sec

Wait 1 sec 'Wait 1 sec with LED off

Loop

Put

Syntax:

Put Line,Column, Character

Explanation:

The Put command writes the given ASCII character code to the location on the LCD.

Example:

```
'A scrolling star for GCBASIC

'Misc Settings

#define SCROLL_DELAY 250 ms

'General hardware configuration

#chip 16F877A, 20

'LCD connection settings

#define LCD_IO 8

#define LCD_DATA_PORT PORTC

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2

'Main routine

For StarPos = 0 To 16
If StarPos = 0 Then
Put 0, 16, 32
Put 0, 0, 42
Else
Put 0, StarPos - 1, 32
Put 0, StarPos, 42
End If
```

Wait SCROLL_DELAY

Next

PWMOff

Syntax:

PWMOff

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command will disable the output of the PWM module on the PIC chip.

Example:

'This program will enable a 76 Khz PWM signal, with a duty cycle

'of 80%. It will emit the signal for 10 seconds, then stop.

```
#define PWM_Freq 76 'Set frequency in KHz
```

```
#define PWM_Duty 80 'Set duty cycle to 80 %
```

```
PWMOn 'Turn on the PWM
```

```
WAIT 10 s 'Wait 10 seconds
```

```
PWMOff 'Turn off the PWM
```

PWMOn

Syntax:

PWMOn

Command Availability:

Only available on PIC microcontrollers with Capture/Compare/PWM (CCP) module.

Explanation:

This command will enable the output of the PWM module on the PIC chip.

Example:

'This program will enable a 76 Khz PWM signal, with a duty cycle

'of 80%. It will emit the signal for 10 seconds, then stop.

```
#define PWM_Freq 76 'Set frequency in KHz
```

```
#define PWM_Duty 80 'Set duty cycle to 80 %
```

```
PWMOn 'Turn on the PWM
```

```
WAIT 10 s 'Wait 10 seconds
```

```
PWMOff 'Turn off the PWM
```

PWMOut

Syntax:

PWMOut *channel, duty cycle, cycles*

Explanation:

This command uses a software PWM routine included in GCBASIC to produce a PWM signal on the selected port of the chip. This routine does NOT require a PWM module on the chip.

channel sets the channel that the PWM is to be generated on. This must have been defined previously by setting the constants SoftPWM1, SoftPWM2, etc. The maximum number of channels available is 4.

duty cycle specifies the PWM duty cycle, and ranges from 0 to 255. 255 corresponds to 100%, 127 to 50%, 63 to 25%, and so on.

cycles is used to set the amount of PWM pulses to supply. This is useful for situations in which a pulse of a specific length is required. The formula for calculating the time taken for one cycle is:

$$T_{\text{CYCLE}} = (28 + 10C)T_{\text{OSC}} + 255\text{PWM_Delay},$$

where C is the number of channels used and T_{OSC} is the length of time taken to execute 1 instruction on the chip (0.2 us on a 20 MHz chip, 1 us on a 4 Mhz chip). PWM_Delay is a length of time specified using the PWM_Delay constant.

Example:

'This program controls the brightness of an LED on PORTB.0 'using the software PWM routine and a potentiometer.

'Select chip model

#chip 16F877A, 20

'Set PWM Port

```
#define PWM_Out1 PORTB.0

'Set port directions

dir SoftPWM1 out 'PWM

dir PORTA.0 in 'Potentiometer

'Main routine

do

PWMOut 1, ReadAD(AN0), 100 '100 cycles is a purely
'arbitrary value here.

Loop
```

Random

Syntax:

```
var = Random
```

Explanation:

The Random function will generate a pseudo-random number between 0 and 255 inclusive.

The numbers generated by Random will follow the same sequence every time, until Randomize is used.

Example:

```
'Set chip model

#chip tiny2313, 1

'Use randomize, with the value on PORTD as the seed
```

```
Randomize PORTD

'Generate random numbers, and output on PORTB

Do

PORTB = Random

Wait 1 s

Loop
```

Randomize

Syntax:

Randomize

Randomize *seed*

Explanation:

Randomize is used to seed the pseudo random number generator, so that it will produce a different sequence of numbers each time it is used.

If no *seed* is specified, then the RANDOMIZE_SEED constant will be used as the seed. If *seed* is specified, then it will be used to seed the generator.

It is important that the seed is different every time that Randomize is used. If the seed is always the same, then the sequence of numbers will always be the same. It is best to use a running timer, an input port, or the analog to digital converter as the source of the seed, since these will normally provide a different value each time the program runs.

Example:

```
'Set chip model
```

```
#chip tiny2313, 1

'Use randomize, with the value on PORTD as the seed

Randomize PORTD

'Generate random numbers, and output on PORTB

Do

    PORTB = Random

    Wait 1 s

Loop
```

ReadAD

Syntax:

```
var = ReadAD (port)
```

Command Availability:

Available on all PIC chips with an analog to digital converter module built in.

Explanation:

ReadAD is a function that can be used to read the built-in analog to digital converter that many microcontroller chips include. *port* should be AN0, AN1, AN2, etc., up to the number of analog inputs available on the chip that is in use. Refer to the datasheet for the microcontroller chip to find the number of ports available. (Note: it's perfectly acceptable to use ADCx on PIC.)

Another function, ReadAD10, is almost identical to ReadAD. The only difference is that it returns a full 10 bit value in a word variable.

Example:

```
#chip 16F819, 8

#config osc = int

'Set the input pin direction
Dir PORTA.0 In

'Loop to take readings until the EEPROM is full
For CurrentAddress = 0 to 255

'Take a reading and log it
EPWrite CurrentAddress, ReadAD(AN0)

'Wait 10 minutes before getting another reading
Wait 10 min

Next
```

ReadTable

Syntax:

ReadTable *TableName*, *Item*, *Output*

Explanation:

The ReadTable command is used to read information from lookup tables. *TableName* is the name of the table that is to be read, *Item* is the line of the table to read, and *Output* is the variable to write the retrieved value in to.

Item is 1 for the first line of the table, 2 for the second, and so on. Item 0 gives the size of the table. Care must be taken to ensure that the program is not told to read beyond the end of the table, or strange effects will be observed.

The type of *Output* should match the type of data stored in the table. For example, if the table contains Word values then *Output* should be a Word variable. If the type does not match, GCBASIC will attempt to convert the value.

Example:

```
'Chip Settings

#chip 16F88, 20

'Hardware Settings

#define LED PORTB.0

Dir LED Out

'Main Routine

ReadTable TimesTwelve, 4, Temp

Set LED Off

If Temp = 48 Then Set LED On

'Lookup table named "TimesTwelve"

Table TimesTwelve

12
```

24

36

48

60

72

84

96

108

120

132

144

End Table

Repeat

Syntax:

```
Repeat times
...
program code
...
End Repeat
```

Explanation:

The Repeat command is ideal for situations where a piece of code needs to be run a set number of times. It uses less memory and runs faster than the For

command, and should be used wherever it is not necessary to count how many times the code has run.

Example:

'This code will flash a green light 6 times.

```
#chip 16F88, 20
```

```
#define LED PORTB.0
```

```
dir LED out
```

```
Repeat 6
```

```
    PulseOut LED, 1 s
```

```
    Wait 1 s
```

```
End Repeat
```

Rotate

Syntax:

Rotate *variable* {Left | Right} [Simple]

Explanation:

The Rotate command will rotate *variable* one bit in a specified direction. The bit shifted will be placed in the Carry bit of the Status register (STATUS.C). STATUS.C acts as a ninth bit of the variable that is being rotated.

When a variable is **rotated right**, the bit in the STATUS.C location is placed into the MSB of the variable being rotated, and the LSB of the variable is placed into STATUS.C location.

When **rotated left** the opposite occurs. The MSB of the variable is shifted to the STATUS.C bit and the LSB of the variable will contain what was previously in the STATUS.C bit location.

This table shows the operation of the Rotate Left command

Command	variable	STATUS.C
Values at start:	b'01110011'	0
Rotate Left	b'11100110'	0
Rotate Left again	b'11001100'	1
Rotate Left third time	b'10011001'	1

As you may notice the STATUS.C bit added a 0 to the rotation. So this will take 9 shifts left to get back to the original value.

SIMPLE option

Many times you want to rotate the variable around like the STATUS.C bit wasn't there so the MSB of the variable fills the LSB of the variable on Rotate Left or the LSB fills the MSB on Rotate Right. That is where the SIMPLE option comes in. It adds a hidden step that shifts the STATUS.C bit twice so the bit moves from one end of the variable to the other.

Command	variable	STATUS.C
Values at start:	b'01110011'	0
Rotate Left	b'11100110'	0
Rotate Left again	b'11001101'	1
Rotate Left third time	b'10011011'	1

Example:

'This program will use Rotate to show a chasing LED.
'8 LEDs should be connected to PORTB, one on each pin.

```
#chip 16F819, 8  
#config osc = int
```

```
'Set port direction
```

```
Dir PORTB Out
```

```
'Set initial state of port (bits 0 and 4 on)
```

```
PORTB = b'00010001'
```

```
'Chase
```

```
Do
```

```
    Rotate PORTB Right Simple
```

```
    Wait 250 ms
```

```
Loop
```

Select

Syntax:

```
Select Case var
```

```
Case value1
```

```
    code1
```

```
Case value2
```

```
    code2
```

```
Case Else
```

```
    code3
```

End Select

Explanation:

The Select Case control structure is used to select and run a particular section of code, based on the value of *var*. If *var* equals *value1* then *code1* will be run. Once *code1* has run, the chip will jump to the End Select command and continue running the program. If none of the other conditions are true, then the code under the Case Else section will be run.

Case Else is optional, and the program will function correctly without it.

If there is only one line of code after the Case, the code may look neater if the line is placed after the Case. This is shown below in the example, for cases 3, 4 and 5.

It is important to note that **only one section of code will be run** when using Select Case.

Example:

'Program to read a value from a potentiometer, and display a

'different word based on the result

```
#chip 18F4550, 20

'LCD connection settings

#define LCD_IO 4

#define LCD_DB4 PORTD.4

#define LCD_DB5 PORTD.5

#define LCD_DB6 PORTD.6

#define LCD_DB7 PORTD.7

#define LCD_RS PORTD.0
```

```

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2

DIR PORTA.0 IN

Do

    Temp = ReadAD(AN0) / 20

    CLS

    Select Case Temp

        Case 0

            Print "None!"

        Case 1

            Print "One"

        Case 2

            Print "Two"

        Case 3: Print "Three"

        Case 4: Print "Four"

        Case 5: Print "Five"

        Case Else

            Print "A lot!"

    End Select

    Wait 250 ms

Loop

```

SerPrint

Syntax:

SerPrint *channel*, *value*

Explanation:

SerPrint is used to send a value over the serial connection. *value* can be a string, word or byte - SerPrint is very similar to Print. *channel* is the serial connection to send data through.

SerPrint will not send any new line characters. If the chip is sending to a terminal, these commands should follow every SerPrint:

SerSend *channel*, 13

SerSend *channel*, 10

Example:

```
'This program will display any values received over the serial
'connection. If "pot" is received, the value of the analog sensor
'will be sent.

'Chip settings
#chip 18F2525, 8
#config Osc = Int

'LCD settings
```

```

#define LCD_IO 4

#define LCD_RS PORTC.7

#define LCD_RW PORTC.6

#define LCD_Enable PORTC.5

#define LCD_DB4 PORTC.4

#define LCD_DB5 PORTC.3

#define LCD_DB6 PORTC.2

#define LCD_DB7 PORTC.1


'Serial settings

#define SerInPort PORTB.6

#define SerOutPort PORTB.7


#define SendAHigh Set SerOutPort Off

#define SendALow Set SerOutPort On

#define RecAHigh SerInPort Off

#define RecALow SerInPort On


'Potentiometer

#define POT_PORT PORTA.0

#define POT_AN AN0


'Set pin directions

Dir SerOutPort Out

Dir SerInPort In

```

```

Dir POT_PORT In

'Create buffer variables to store received messages

Dim Buffer As String
Dim OldBuffer As String
BufferSize = 0

'Set up serial connection
InitSer 1, r9600, 1 + WaitForStart, 8, 1, none, invert

'Show test messages
Print "Serial Tester"
Wait 1 s
SerPrint 1, "GCBASIC RS232 Test"
SerSend 1, 13
SerSend 1, 10
Wait 1 s

'Main loop
Do
    'Get a byte from the terminal
    SerReceive 1, Temp

    'If Enter key was pressed, deal with buffer contents
    If Temp = 13 Then

```

```

Buffer(0) = BufferSize

'Try to execute commands in buffer

If Not ExecCommand (Buffer) Then

    'Show message on bottom line, last message on top.

    CLS

    Print OldBuffer

    Locate 1, 0

    Print Buffer

    'Store the message for next time

    OldBuffer = Buffer

End If

BufferSize = 0

End If

'Backspace code, delete last character in buffer

If Temp = 8 Then

    If BufferSize > 0 Then BufferSize -= 1

End If

'Received ASCII code between 32 and 127, add to buffer

If Temp >= 32 And Temp <= 127 Then

    BufferSize += 1

    Buffer(BufferSize) = Temp

```

```

    End If

Loop

'Takes a sensor reading and sends it to terminal
Sub SendSensorReading

    SerPrint 1, "Sensor Reading: "

    SerPrint 1, ReadAD10(POT_AN)

    SerSend 1, 13

    SerSend 1, 10

End Sub

'Will check the buffer for a command
'If command found, run it and return true
'If not, return false
Function ExecCommand (CmdIn As String)

    ExecCommand = False

    'If received command is "pot", show potentiometer value

    If CmdIn = "pot" Then

        SendSensorReading

        ExecCommand = True

    End If

End Function

```

SerReceive

Syntax:

SerReceive *channel*, *output*

Explanation:

This command will read a byte from the RS232 channel given by *channel* according to the rules set using InitSer, and store the received byte in the variable *output*.

Example:

'This program will read a byte from PORTB.2, and set the LED on if
'the byte is more than 50. This can be used with the SerSend
'example program.

```
#chip 16F88, 8  
  
#config Osc = Int  
  
#define RecAHigh PORTB.2 ON  
#define RecALow PORTB.2 OFF  
#define LED PORTB.0  
  
Dir PORTB.0 Out  
Dir PORTB.2 In
```

```
InitSer 1, r9600, 1 + WaitForStart, 8, 1, none, normal
```

```
Do
```

```
    SerReceive 1, Temp
```

```
    If Temp <= 50 Then Set LED Off
```

```
    If Temp > 50 Then Set LED On
```

```
Loop
```

SerSend

Syntax:

```
SerSend channel, data
```

Explanation:

This command will send a byte given by *data* using the RS232 channel referred to as *channel* according to the rules set using InitSer.

Example:

```
'This program will send a byte using PORTB.2, the value of which  
'depends on whether a button is pressed. This can be used with the  
'example for SerReceive.
```

```
#chip 16F819, 8
```

```
#config Osc = Int
```

```

#define SendAHigh Set PORTB.2 ON

#define SendALow Set PORTB.2 OFF

#define Button PORTA.0

Dir Button In

Dir PORTB.2 Out

InitSer 1, r9600, 1+WaitForStart, 8, 1, none, normal

Do

    If Button = On Then Temp = 100

    If Button = Off Then Temp = 0

    SerSend 1, Temp

    Wait 100 ms

Loop

```

Set

Syntax:

Set *variable.bit* {On | Off}

Explanation:

The purpose of the Set command is to turn individual bits on and off. The Set command is most useful for controlling output ports, but can also be used to set variables.

Often when controlling output ports, Set is used in conjunction with constants. This makes it easier to adapt the program for a new circuit later.

Example:

'Blink LED sample program for GCBASIC

'Controls an LED on PORTB bit 0.

'Set chip model and config options

#chip 16F84A, 20

'Set a constant to represent the output port

#define LED PORTB.0

'Set pin direction

Dir LED Out

'Main routine

Do

Set LED On

Wait 1 sec

Set LED OFF

Wait 1 sec

Loop

ShortTone

Syntax:

ShortTone Frequency, Duration

Explanation:

This command will produce the specified tone for the specified duration. Frequency is measured in units of 10 Hz, and Duration is in 1 ms units.

Please note that this command may not produce the exact frequency specified. While it is accurate enough for error beeps and small pieces of monophonic music, it should not be used for anything that requires a highly precise frequency.

Example:

'Sample program to produce a tone on PORTB bit 1, based on the
'reading of an LDR on AN0 (usually PORTA bit 0).

```
#chip 16F88, 20
```

```
#define SoundOut PORTB.1
```

```
Dir PORTA.0 In
```

```
Do
```

```
    ShortTone ReadAD(AN0), 100
```

```
Loop
```

SPIMode

Syntax:

SPIMode *Mode*

Command Availability:

Available on PIC microcontrollers with Synchronous Serial Port (SSP) module.

Explanation:

SPIMode sets the mode of the SPI module within the PIC chip. These are the possible SPI Modes:

Mode Name	Description
MasterSlow	Master mode, SPI clock is 1/64 of the frequency of the PIC.
Master	Master mode, SPI clock is 1/16 of the frequency of the PIC.
MasterFast	Master mode, SPI clock is 1/4 of the frequency of the PIC.
Slave	Slave mode
SlaveSS	Slave mode, with the Slave Select pin enabled.

SPITransfer

Syntax:

SPITransfer *tx*, *rx*

Command Availability:

Available on PIC microcontrollers with Synchronous Serial Port (SSP) module.

Explanation:

This command simultaneously sends and receives a byte of data using the SPI protocol. It behaves differently depending on whether the PIC has been set to act as a master or a slave.

When operating as a master, SPITransfer will initiate a transfer. The data in *tx* will be sent to the slave, whilst the byte that is buffered in the slave will be read into *rx*.

In slave mode, the SPITransfer command will pause the program until a transfer is initiated by the master. At this point, it will send the data in *tx* whilst reading the transmission from the master into the *rx* variable.

Example:

There are two example programs for this command - one to run on the slave PIC, and one on the master. A reading is taken from a sensor on the slave, and sent across to the master which shows the data on its LCD screen.

Slave Program:

'Select chip model and configuration

#chip 16F88, 20

#config MCLR_OFF

'Set directions of SPI pins

dir PORTB.2 out

dir PORTB.1 in

dir PORTB.4 in

'Set direction of analogue pin

dir PORTA.0 in

'Set SPI mode to slave

SPIMode Slave

'Allow other PIC to initialise LCD

Wait 1 sec

'Main loop - takes a reading, and then waits to send it across.

do

'Note that rx is 0 - this is because no data is to be received.

SPITransfer ReadAD(AN0), 0

loop

Master Program:

'General hardware configuration

#chip 16F877A, 20

'LCD connection settings

#define LCD_IO 8

#define LCD_DATA_PORT PORTC

#define LCD_RS PORTD.0

#define LCD_RW PORTD.1

#define LCD_Enable PORTD.2

'Set SPI pin directions

dir PORTC.5 out

dir PORTC.4 in

dir PORTC.3 out

'Set SPI Mode to master, with fast clock

SPIMode MasterFast

'Main Loop

do

'Read a byte from the slave

'No data to send, so tx is 0

SPITransfer 0, Temp

'Display data

if Temp > 0 then

CLS

Print "Light: "

LCDInt Temp

Temp = 0

end if

'Wait to allow time for the LCD to show the given value

wait 100 ms

loop

StartTimer

Syntax:

StartTimer *TimerNo*

Command Availability:

Available on all PIC microcontrollers with built in timer modules.

Explanation:

StartTimer is used to start the specified timer. Note that it cannot be used to start Timer 0 on a PIC - this timer always runs.

StopTimer

Syntax:

StopTimer *TimerNo*

Command Availability:

Available on all PIC microcontrollers with built in timer modules.

Explanation:

StopTimer is used to stop the specified timer. Note that it cannot be used to stop Timer 0 on a PIC - this timer always runs.

Example:

Please refer to the InitTimer1 article for an example.

Tone

Syntax:

Tone *Frequency, Duration*

Explanation:

This command will produce the specified tone for the specified duration. *Frequency* is measured in Hz, and *Duration* is in 10 ms units.

Please note that this command may not produce the exact frequency specified. While it is accurate enough for error beeps and small pieces of monophonic music, it should not be used for anything that requires a highly precise frequency.

Example:

```
'Sample program to produce a constant A note (440 Hz)

'on PORTB bit 1.

#chip 16F877A, 20

#define SoundOut PORTB.1

Do

    Tone 440, 1000

Loop
```

Wait	
-------------	--

Syntax:

- Fixed Length Delay:** Wait *time units*
- Conditional Delay:** Wait {While | Until} *condition*

Explanation:

The Wait command will cause the program to wait for either a specified amount of time (such as 1 second), or while/until a condition is true.

When using the fixed-length delay, there is a variety of units that are available:

Unit	Length of unit	Delay range
------	----------------	-------------

us	1 microsecond	1 us - 65535 us
10us	10 microseconds	10 us - 2.55 ms
ms	1 millisecond	1 ms - 65535 ms
10ms	10 milliseconds	10 ms - 2.55 s
s	1 second	1 s - 255 s
m	1 minute	1 min - 255 min
h	1 hour	1 hour - 255 hours

At one stage, GCBASIC variables could not hold more than 255. The 10us and 10ms units were added as a way to work around this limit. There is now no such limit (Wait 1000 ms will work for example), so these are not really needed. However, you may see them in some older examples or programs, and the 10us units are sometimes the shortest delay that will work accurately.

Example:

'This code will wait until a button is pressed, then it will flash
'a light every half a second and produce a 440 Hz tone.

```
#chip 16F819, 8
#config osc = int

#define BUTTON PORTB.0
#define SPEAKER PORTB.1
#define LIGHT PORTB.2

Dir BUTTON In
Dir SPEAKER Out
Dir LIGHT Out
```

'Assumes Button switches on when pressed

Wait Until BUTTON = 1

Wait Until BUTTON = 0

Do

'Flash the light

Set LIGHT On

Wait 500 ms

Set LIGHT Off

'Produce the tone

'440 Hz = 880 changes = tone on for 1.14 ms

Repeat 440

PulseOut SPEAKER, 1140 us

Wait 114 10us 'Wait for 114 x 10 us (1.14 ms)

End Repeat

Loop

Sample Version - Do Not Copy

6.0 Sample Projects

Here are a few sample projects to help you get started with Great Cow Basic CHIPINO. They are designed to work with the CHIPINO Demo-Shield but you can build the circuits separately if you know how to do that.



CHIPINO Demo-Shield

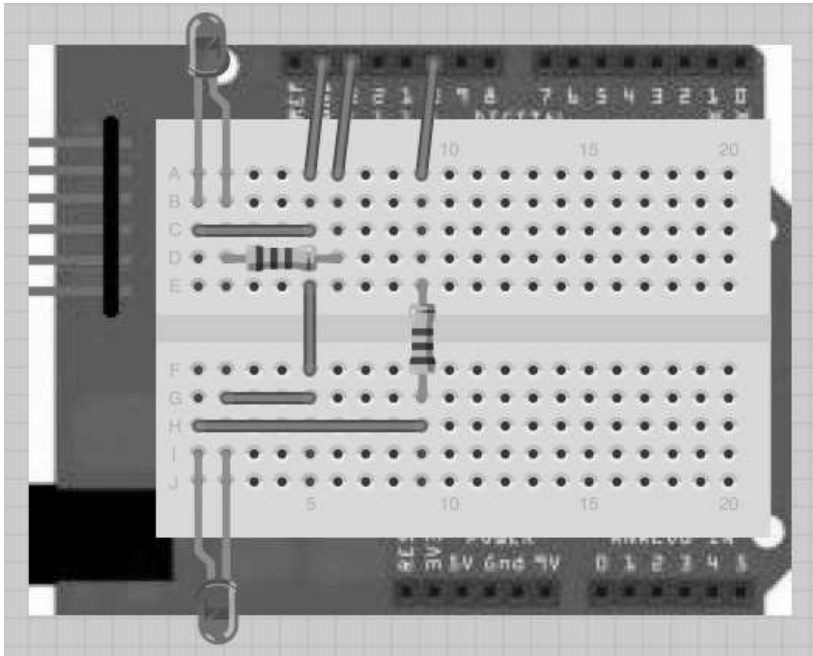
Project 1 - Train Crossing

Flashing an LED is a great place to start but let's put it to some practical use. Have you ever stopped at a train crossing and seen the red lights flash back and forth to indicate a train is coming? This can be accomplished real easy with two LEDs. This project can also be built into a model train railroad crossing sign to make it a little more realistic.

Hardware

There are two LEDs to control. The LEDs are driven on by a high signal on the digital pin that drives them. A low shuts them off. A resistor is placed in series with the LEDs to limit the current. A 1k ohm resistor works fine.

The left LED is connected to the digital pin number 13 on the CHIPINO. The right LED is connected to pin 10. A series resistor is used to limit the current and any value from 220 ohms to 1k ohms can be used. Both LEDs are red on the CHIPINO demo shield.



Drawn with Fritzing.org
Figure 6-1: Train Crossing Project

Software

```
'Train.gcb
'A program to flash two red LEDs like a
'train crossing on Digital pins 10 and 13
'(red LEDs on Demo-Shield)

'Chip model
#chip 16F886, 4
#include <chipino.h>      'Defines CHIPINO setup

'Main routine
Start:
'Turn one LED on, the other off
set D10 on  'Left LED on
set D13 off 'Right LED off
wait 1 sec
```

```
'Now toggle the LEDs
Set D10 OFF      'Left LED off
Set D13 ON 'right LED on
wait 1 sec
```

```
'Jump back to the start of the program
goto Start
```

How it Works

The first section is just a comment header to describes the project. Each line is a comment line since it starts with an ' apostrophe.

```
'Train.gcb
'A program to flash two red LEDs like a
'train crossing on Digital pins 10 and 13
'(red LEDs on Demo-Shield)
```

The PIC16F886 in the CHIPINO is selected and then a header file with all the nicknames D0-D13 and A0-A5.

```
'Chip model
#chip 16F886, 4
#include <chipino.h>      'Defines CHIPINO setup
```

The main loop controls the two separate LEDs by controlling the digital pins they are connected to. Setting pin D13 off and pin D10 on lights the left LED. A wait command for 1 second creates the flash rate. The second section does the opposite and lights the right LED on pin D13 and shuts off the left LED on pin D10. Another wait of 1 second is added.

```
'Main routine
Start:
'Turn one LED on, the other off
set D10 on 'Left LED on
set D13 off 'Right LED off
wait 1 sec
```

```
'Now toggle the LEDs
```

```
Set D10 OFF      'Left LED off  
Set D13 ON 'right LED on  
wait 1 sec
```

The program stays in a continuous loop with the goto Start command line creating an alternating lighting of the LEDs to create the train crossing display.

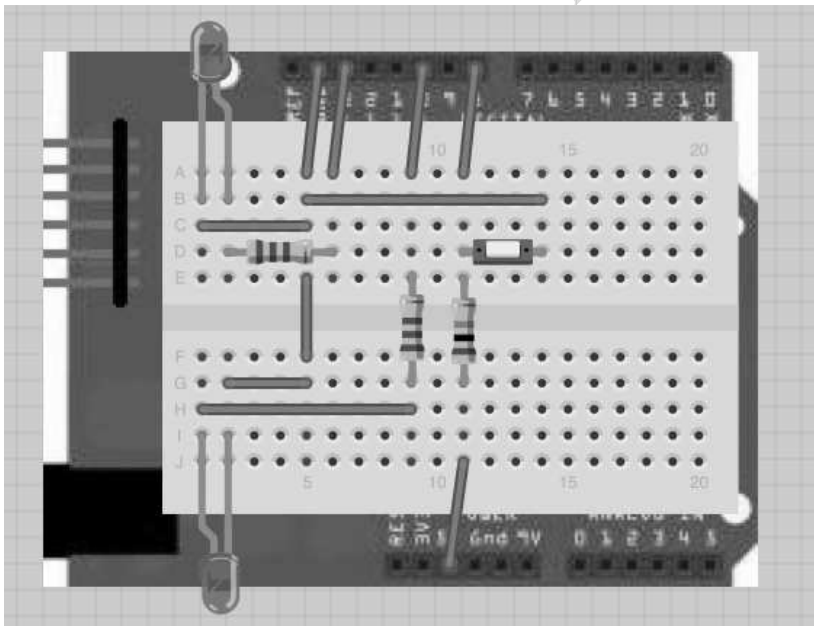
```
'Jump back to the start of the program  
goto Start
```

Sample Version - Do Not Copy

Project 2 - Sensing a Switch

On many projects you will need some kind of human interface to control the operation. A momentary push button switch is a very common way to do that. It can start and stop the operation or it could speed up or slow down what the microcontroller is controlling. In order to do that though the software needs to recognize that a switch was pressed. This project shows a simple method of sensing a momentary push button switch.

The software will have to monitor the switch continuously as part of the main loop of code and then respond. In this project the software will light one LED until the switch is pressed at which point the LED will shut off and a second LED with light. As long as the switch is pressed the first LED will stay off and the second LED will stay on. As soon as the switch is released the first LED will once again light up and the second LED will shut off. The completed project is shown in Figure 6-2.



Drawn with Fritzing.org

Figure 6-2: Switch Control

Hardware

The hardware uses the same two Red LED connections as the train crossing project in Chapter 3. The addition of the switch is shown in Figure 6-2. The switch is wired as a low side switch meaning the circuit has a pull-up resistor to 5 volts so the input the micro is high when the switch is not pressed and low when the switch is pressed. This is known as a low side switch. If the parts were reversed and the switch was connected to 5 volts and the resistor to ground then it would be a high side switch.

The software will change pin 8 to an input and then test pin 8 to see if it changes to low indicating the switch has been pressed. The LEDs are connected to pin 10 and pin 13 through 1k resistors.

Software

```
'switches.gcb
'Monitor two switches and light LED when pressed.
'Other LED lit when idle
'Setup for Demo-Shield
'D8 switch controls D10 LED
```

```
'Chip Settings
#chip 16F886,4
#include <chipino.h>
DIR D8 in 'Set D8 switch pin to input
DIR D10 out 'Set D10 LED pin to output
DIR D13 out 'Set D13 LED pin to output
```

```
Start:
If D8=0 Then 'If Left Switch Pressed
  set D10 on 'D10 LED on
  set D13 off 'D13 LED off
Else
  set D13 on 'D13 LED on
  set D10 off 'D10 LED off
End If
Goto Start
```

How it Works

The first section is just a comment header to describes the project. Each line is a comment line since it starts with an ' apostrophe.

```
'switches.gcb
'Monitor two switches and light LED when pressed.
```

```
'Other LED lit when idle
'Setup for Demo-Shield
'D8 switch controls D10 LED
```

The PIC16F886 in the CHIPINO is selected and then a header file with all the nicknames D0-D13 and A0-A5.

```
'Chip model
#chip 16F886, 4
#include <chipino.h>      'Defines CHIPINO setup
```

The port direction for each pin is defined. The LEDs don't need this since the GCB compiler automatically sets pins to outputs but I added it here to show how to setup both an input and output pin.

```
DIR D8 in   'Set D8 switch pin to input
DIR D10 out 'Set D10 LED pin to output
DIR D13 out 'Set D13 LED pin to output
```

Now we enter the main loop. An If Else Endif command is used. The first section tests if the D8 pin is low indicating the switch has been pressed. If it is low then the D10 LED is lit and the D13 is shut off. If D8 pin is not low then the Else section is performed and D13 LED is let and D10 is shut off. The End if indicates the end of the If Else command.

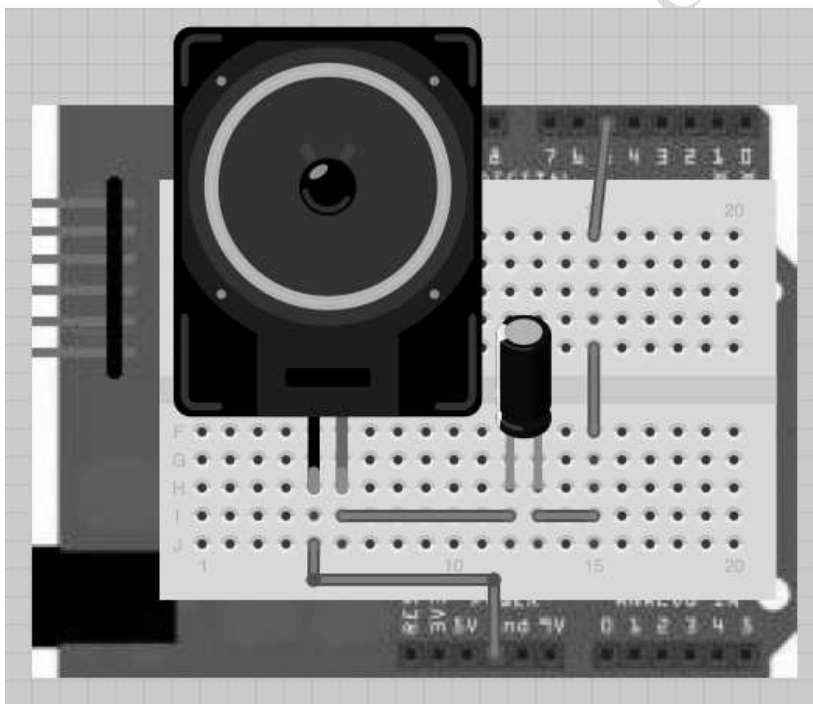
```
Start:
If D8=0 Then 'If Left Switch Pressed
  set D10 on 'D10 LED on
  set D13 off 'D13 LED off
Else
  set D13 on 'D13 LED on
  set D10 off 'D10 LED off
End If
```

A Goto Start command line loops back to the label Start to make the program continuously monitor the switch input.

```
Goto Start
```

Project 3 – Creating Sound

Creating sound through a speaker can be very useful for a lot of projects. You can create a sound when a switch is pressed or sound an alarm when the temperature drops. In this project I'll show you how to make a tone through a speaker using a digital pin. I've selected a PWM pin because the demo-shield is wired like that but this can be run on any digital pin. The Tone command in Great Cow Basic will be used to make this a very short program to write. This project will create the sound of a falling object similar to what you might hear on an old video game.



Drawn with Fritzing.org

Figure 6-3: Creating a Tone

Hardware

The CHIPINO D5 pin will produce a square wave because it's a digital pin driving the speaker. The square wave can be converted into a semi-rounded signal to work better with the speaker by placing a 10 uf capacitor in series between pin 5 and the speaker's positive lead. The other side of the speaker is then grounded. An 8-ohm speaker works fine but a more common approach is a piezo speaker. A Piezo speaker from Jameco.com under part number DBX05-PN will work well.

The setup is shown in Figure 6-3. The right side of the capacitor is the positive side and that connects to the digital pin D5. The speaker positive is also on the right side and connects to the negative side of the capacitor. The speaker is then grounded to the ground pin on the CHIPINO power header.

Software

```
'tone.gcb
'This program produces a falling sound
'on Digital Pin 5 connected to Piezo
'speaker on Demo-Shield.

#chip 16F886, 4
#include <chipino.h>
DIM SND as word 'Create a word variable

#define SoundOut D5
Start:
For SND = 2000 to 200 step -5
    Tone SND, 1 'Tone with 10 msec delay
Next
goto start
```

How it Works

The first section is just a comment header to describes the project. Each line is a comment line since it starts with an ' apostrophe.

```
'tone.gcb
'This program produces a falling sound
'on Digital Pin 5 connected to Piezo
'speaker on Demo-Shield.
```

The PIC16F886 in the CHIPINO is selected and then a header file with all the nicknames D0-D13 and A0-A5.

```
#chip 16F886, 4  
#include <chipino.h>
```

The tone command will use a value larger than 255 so we need to define a word variable we named SND (short for sound). Word variables will hold a number up to 65,535. GCB automatically creates byte variables when you use a variable but for larger than a byte size you need to define it using the DIM command line.

DIM SND as word 'Create a word variable

Now we enter the main loop. A For Next command is used to create the sound producing loop. It starts by setting the SND variable to 2000 and then steps down by 5 each loop until SND equals 200. Then Tone command is used to produce the tone in the speaker. It produces a frequency in the speaker equal to the value of the SND variable. It starts with a 2000 hertz tone and slowly drops down to 200 hertz. The speed is set by how long each tone lasts and then is controlled by the value in the Tone command line. In this case its set to produce a tone for 10 milliseconds each.

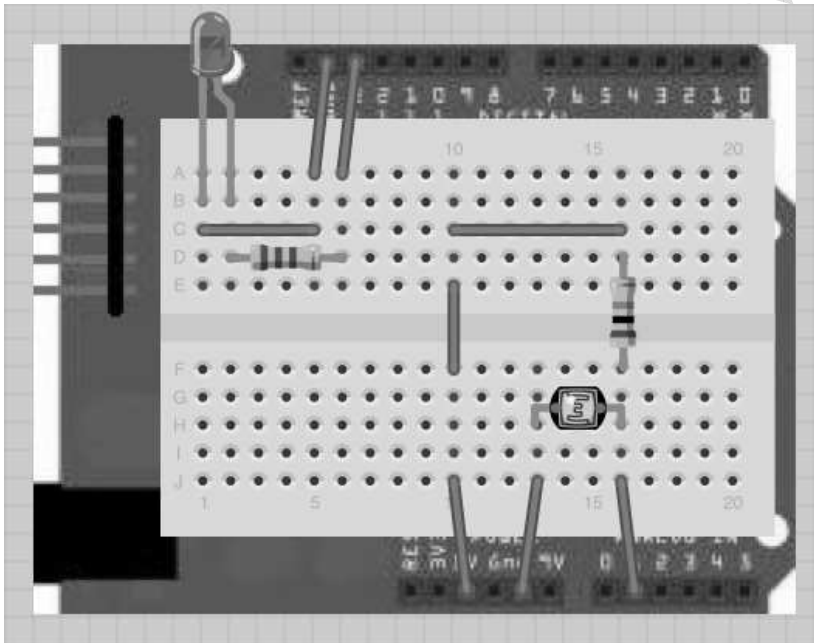
```
Start:  
For SND = 2000 to 200 step -5  
    Tone SND, 1 'Tone with 10 msec delay  
Next
```

The For Next loop automatically continuously loops but we add a Goto Start at the bottom just in case the software gets lost and has to get back to the beginning.

Goto Start

Project 4 - Sensing Light

We can use a photo resistor to sense light using an analog pin. A photo resistor changes resistance as it is exposed to light. By putting a pull-up resistor connected to 5v in series with the photo resistor, changing light will create a variable voltage. The variable voltage can then be monitored by an Analog pin on the CHIPINO. In this case though we'll just light an LED when it's dark and turn it off when there is light.



Drawn with Fritzing.org

Figure 6-4: Creating a Tone

Hardware

One side of the light sensor is connected to ground. The other connects to the resistor and also the analog pin 1. The pull-up resistor is a 10k connected to 5v. You may have to adjust the resistor for your light sensor but in most cases a 10k will work fine. In the dark the photo resistor has a high resistance and in the light it has a low resistance. Therefore in the dark the voltage will be high and in the light the voltage will be low.

The LED controlled by the sensor is connected to pin 10 of the digital port. A 220 ohm series resistor connected to the anode and the cathode connected to ground completes the circuit.

Software

```
'light.gcb
'This program monitors a photoresistor connected
' to the AN1 pin. If it's in the dark it will
' light the LED on D13. If it's in the light
' then LED is shutoff.
'Written for the CHIPINO Demo-Shield.
```

```
'Chip Settings
#chip 16F886, 4
#include <chipino.h>
Dir A1 In    'Make Analog 1 pin input
Dir D13 out  'Make D15 pin output
```

```
Start:
Do
'Light LED if in the dark
If ReadAD(A1) < 150 Then
  set D13 on 'Sensor is Dark
else
  set D13 off 'Bright light on sensor
end if
```

```
Loop
```

How it Works

The first section is just a comment header to describes the project. Each line is a comment line since it starts with an ' apostrophe.

```
'light.gcb
'This program monitors a photoresistor connected
' to the AN1 pin. If it's in the dark it will
' light the LED on D13. If it's in the light
' then LED is shutoff.
'Written for the CHIPINO Demo-Shield.
```

The PIC16F886 in the CHIPINO is selected and then a header file with all the nicknames D0-D13 and A0-A5.

```
#chip 16F886, 4
#include <chipino.h>
```

The A1 analog pin is set to an input to read the light sensor voltage. The port direction for the D13 pin is also defined as an output. The Digital pins don't need this since the GCB compiler automatically sets pins to outputs but I added it here to show how to setup the output pin if you wanted to define it.

```
Dir A1 In    'Make Analog 1 pin input  
Dir D13 out  'Make D15 pin output
```

Now we enter the main loop. In this case it's an actual Do Loop command instead of the Lable Goto Lable I've used previously. The Do Loop will run whatever is between Do and Loop continuously. I added a Start: label just to indicate where the main loop starts but it really serves no purpose to the program.

The ReadAD command is used to read the voltage on the A1 pin of the CHIPINO. It will use the internal Analog to Digital Converter (ADC) built into the analog pins of the CHIPINO to convert the voltage at A1 to a value of 0 to 255. We can then compare that to a value, in this case 150, to determine if the LED should light or not.

If the value converted by the ReadAD command is less than 150 (dark) then the D13 LED is turned on. If the value is greater than 150 (light) then the D13 LED is turned off.

```
Start:  
Do  
'Light LED if in the dark  
If ReadAD(A1) < 150 Then  
  set D13 on 'Sensor is Dark  
  else  
  set D13 off 'Bright light on sensor  
end if
```

```
Loop
```

Sample Version - Do Not Copy

Appendix A – PICKit 2 GUI Features

The PICKit 2 Programmer from Microchip is an open source hardware programmer. The Great Cow Basic CHIPINO USB Programming cable is actually a PICKit 2 Programmer shrunk down to a cable. The PICKit 2 / USB Programming Cable can do more than program the CHIPINO module. It can also receive signals for a terminal program called the UART Tool and display signals in digital mode or in a logic analyzer graphic mode with the Logic Tool. This appendix will cover those features.

You can launch the PICKit 2 GUI from the GCB IDE toolbar by clicking on the PICKit2 Icon.



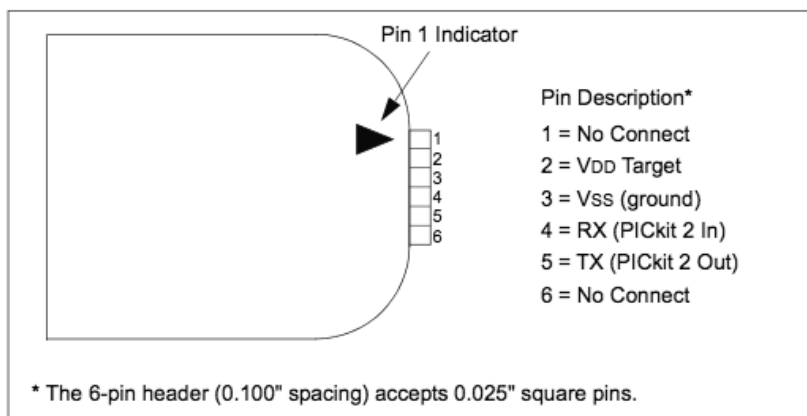
Note: When using the PICKit 2 GUI features, you can't program the CHIPINO using the built in PICKit 2 programmer application associated with the Blue Arrow compile and program icon. You'll need to shut down the PICKit 2 GUI, then reprogram the CHIPINO, then relaunch the PICKit 2 GUI.

UART Tool

The PICKit 2 Programmer GUI includes the UART Tool feature. This feature allows the CHIPINO USB Programmmer cable to be used as a serial UART terminal interface for communicating with a PIC microcontroller. Potential uses include:

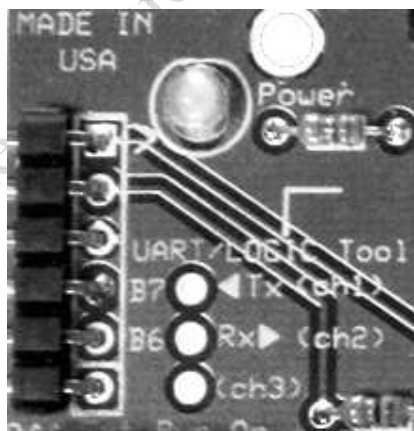
- Displaying debug text output from the microcontroller
- Logging microcontroller data to a text file
- Developing and debugging a microcontroller UART interface
- Interfacing with and sending commands to the microcontroller during development

The tool supports full duplex asynchronous serial communications from 150 to 38400 baud, including custom non-standard baud rates. The PICKit 2 connects via the CHIPINO ICSP connector directly to microcontroller pin RB7 (PICKit 2 RX) and pin RB6 (PICKit 2 TX) at logic levels. No transceiver is needed.



Connecting the UART Tool

The CHIPINO pins B6 and B7 are not connected to the headers. These pins have holes located near the programming connection. You can solder a three pin header there to make connecting easier. Any pin you want to send data through can be connected to the B7 pin to send data to the PICkit 2 Terminal Software. Any pin you want to receive data from the PICkit 2 terminal can be connected to the B6 pin.

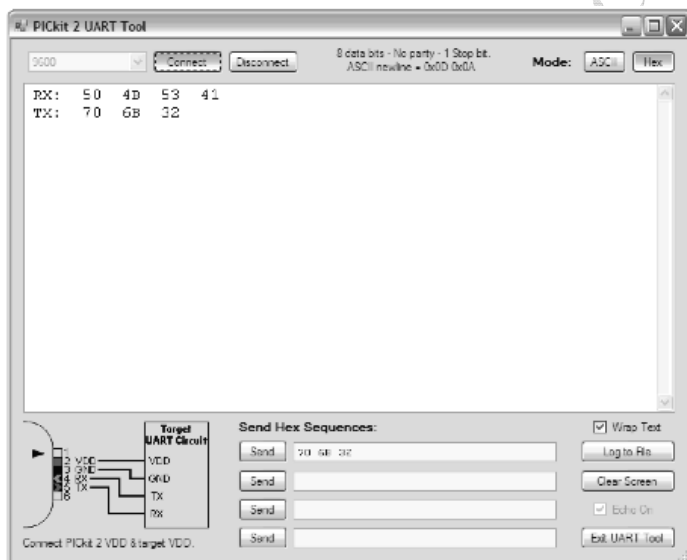


An easier way though is to just route those signals through software to the B7 and B6 pins. This way you have full access to the CHIPINO header pins for signals and use B7 and B6 for communication.

UART Tool Window

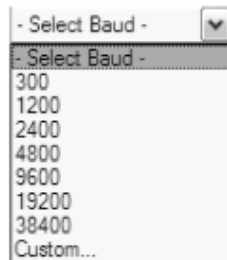
To open the PICKit 2 UART Tool window, select *Tools>UART Tool*. The PICKit 2 Programmer window will open and the PICKit 2 Programmer application window will close, as the programmer cannot be used while the UART Tool is active, and vice versa.

The UART Tool has two modes: ASCII and HEX. The current mode is selected by the buttons on the upper right hand of the display. The button corresponding to the active mode will be displayed depressed. The mode selection affects how serial data is displayed and entered in the window, and the modes are explained in the next two sections.



Setting the Baud Rate and Connecting

The baud rate may be selected from the combo box in the upper left corner of the UART Tool window. Several common baud rates are listed, and by selecting Custom any baud rate from 150 to 38400 in 1 baud increments may be used.



ASCII Mode

When ASCII mode is selected, serial bytes received from the CHIPINO are displayed as ASCII characters in the main window terminal display. All bytes are displayed consecutively. To display a new line, the CHIPINO must transmit the character values 0x0D (carriage return) and 0x0A (line feed) in sequence. If one of these values is sent, or they are sent in reverse order, the character(s) will appear as an unprintable character (box) in the display.

Bytes may be transmitted in three ways:

1. Click on the terminal display to select it. Any characters typed on the PC keyboard will be immediately transmitted out of PICKit 2.
2. Right click on the terminal display and select Paste from the pop-up menu to paste any previous copied or cut text. <Ctrl> + <v> may also be used. Any pasted data will immediately be transmitted out of PICKit 2.
3. Use the String Macros at the bottom of the window.

The String Macros allow up to four strings of characters to be entered. Each string can be up to 60 characters long. When Send is clicked, the entire string will be transmitted in sequence out of the PICKit 2. The String Macros strings are saved even when the UART Tool is not in use.

When checked, the Append CR+LF checkbox above the string macro boxes will automatically transmit the carriage return (0x0D) and line feed (0x0A) characters at the end of a string. When unchecked, nothing is transmitted after the string characters.

No transmitted characters will be displayed if the Echo On checkbox is unchecked. Check this box to display transmitted characters along with received characters in the terminal display.

Text in the terminal display may be selected (highlighted) with the cursor and copied into the clipboard by right clicking and selecting Copy or pressing <Ctrl> + <C>.

Hex Mode

The UART Tool Hex mode displays the hex values of bytes received from the CHIPINO in the terminal display. A line of bytes received by the UART Tool is preceded with the text "RX:". A line of bytes transmitted by the UART Tool is preceded with the text "TX:".

In Hex mode, bytes may only be transmitted one way: by typing a sequence of one or more hex values in one of the four Send Hex Sequences boxes. To send a sequence, click Send next to it. A hex sequence may contain from 1 to 48 bytes.

The Echo On checkbox has no function in Hex mode.

Data in the terminal display may be selected (highlighted) with the cursor and copied into the clipboard by right clicking and selecting Copy or pressing <Ctrl> + <C>.

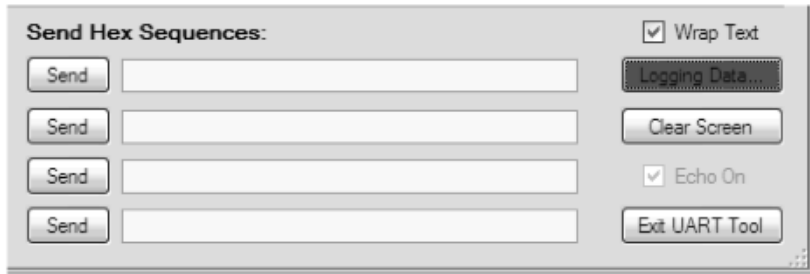
Wrap Text

This checkbox affects both modes, ASCII and HEX. When checked, a displayed line will automatically wrap at the right edge of the terminal display. If unchecked, then a received line will not wrap and a horizontal scroll bar will appear in the terminal display to allow the entire line to be viewed.

The terminal display will keep the last 200 lines of received data in the buffer. The vertical scroll bar may be used to view previous data or text.

Log to File

The Log to File button allows all text and data to be saved to a text file as it appears on the display. Click Log to File to bring up a Save File dialog. Specify the file location and name and click Save. While the terminal display is being saved to the log file, the Log to File button will turn green and display the text "Logging Data".



To stop logging data from the display and close the log file, click Logging Data.

Clear Screen

Click the Clear Screen button to clear all text or data from the terminal display window. If data is being logged to a file, clearing the screen will not affect the log file.

Exit UART Tool

Click Exit UART Tool to close the UART Tool window and return to the PICKit 2 Programmer application main window. The UART Tool terminal display data and settings will be preserved during any programming operations until the UART Tool is re-entered.

If the PICKit 2 application is closed, the UART Tool settings, including String Macros and Hex Sequences, will be saved and restored for the next time the UART Tool is used. The terminal display buffer text and data is not saved when the application is closed.

Logic Tool

The PICKit 2 Logic Tool allows the PICKit 2 ICSP connector pins to be used for stimulating and probing digital signals in the CHIPINO, and as a simple 3 channel logic analyzer. The Logic Tool is opened by selecting Tools > Logic Tool in the main PICKit 2 application window.

The Logic Tool has two operating modes:

1) Logic I/O Mode

This mode is useful for triggering inputs to the CHIPINO, and can monitor digital signals to display their state. In essence, it provides an alternative for wiring buttons and LEDs to pins or signals while debugging or developing I/O functions.

2) Logic Analyzer Mode

The Analyzer mode can display waveforms of up to 3 digital signals, and trigger on specific events such as a rising edge on one signal when another signal is at a logic high level. This may be very useful for debugging serial communication buses such as UART, SPI, and I2C. It is also very applicable to monitoring the behavior of general microcontroller I/O.

Logic I/O Mode

The PICkit 2 Logic Tool "Logic I/O" mode is the default mode when the Logic Tool is first opened. It allows simple stimulus and monitoring of digital signals. The Logic Tool mode is set by the two buttons in the upper right of the Logic Tool window.

The 6-pin CHIPINO programming connector has 4 signal pins that can be used to inject a digital signal into a circuit or display the state of a digital signal from a circuit. The remaining two pins are dedicated for Vdd and Ground connections.

The 6 pins can function as follows in Logic I/O mode:

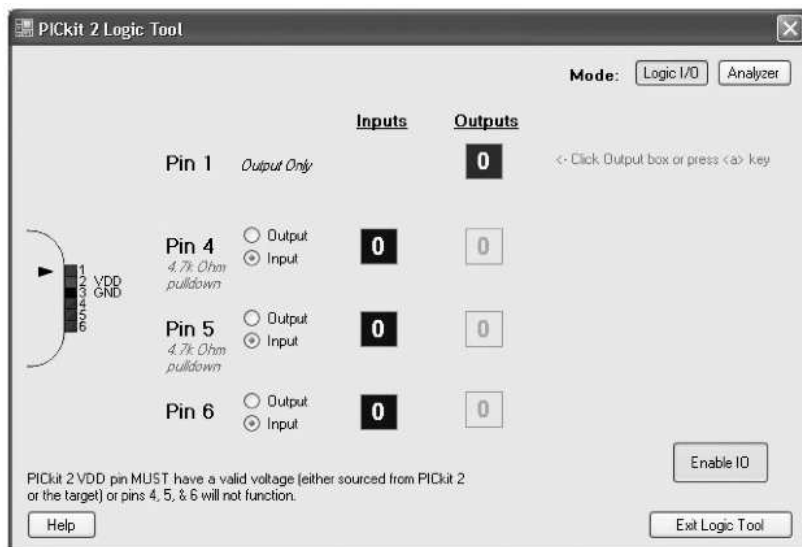
Pin	Function
1	Digital Output Only
2	VDD
3	GND
4	(RB7) Digital Output or Digital Input
5	(RB6) Digital Output or Digital Input
6	AUX Digital Output or Digital Input

Configuring the Logic Tool Logic I/O

To use the Logic I/O mode the Logic I/O mode button on the upper right of the logic window must be depressed. The four pins used for Logic I/O digital signals (pins 1, 4, 5, & 6) will remain tri-stated (inactive) until the Enable IO button is pressed. Once the IO is enabled, it becomes active and can be configured. If no

valid voltage is detected on the VDD pin when clicking Enable IO a dialog will pop up to alert the user, and the pins will remain disabled.

When the pins are enabled, the pin directions and output states can be configured.



Setting Pin Direction

Pins 4, 5, & 6 may be configured as Outputs (output a digital signal) or Inputs (monitor a digital signal state connected to the pin). Pin 1 is only available as an Output.

Click the radio buttons next to the Pin # to set the pin as an Output or Input. When the pin is an Input, the connected signal state is displayed in the blue "Inputs" box.

Note: Pin 4 and Pin 5 have a 4.7k Ohm pulldown resistor internal to the PICKit 2. This resistor is necessary for the PICKit 2 debugger functions, but note that this pulldown resistor will affect any digital signal it is connected to. Generally, this is only an issue when using Pin 4 or Pin 5 as an input.

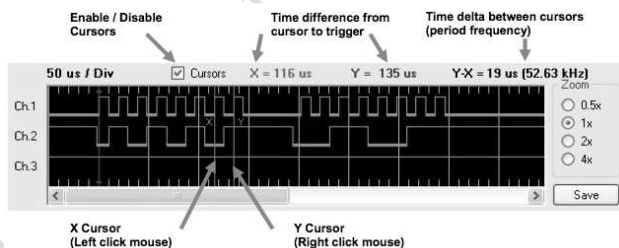
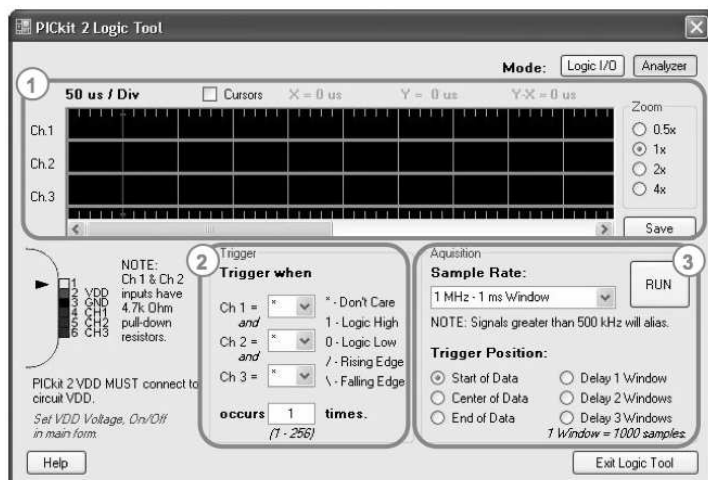
When a pin is selected as an Output, the pin will drive the logic level shown in the read "Outputs" box. Toggle the output state by clicking on the Output state box. Alternatively, a keyboard shortcut key can be used for each pin to toggle the output. The shortcut keys are:

Pin	Shortcut Key
1	<A>
4	<S>
5	<D>
6	<F>

Sample Version - Do Not Copy

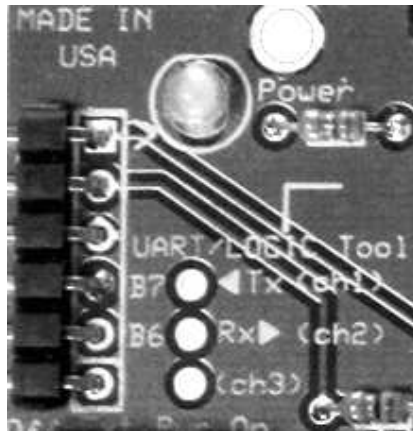
Logic Analyzer Mode

The Analyzer mode of the PICKit 2 Logic Tool enables using the PICKit 2 GUI as a simple three channel logic analyzer to capture, view, and measure the digital waveforms of up to three signals.



Connecting the Analyzer

The CHIPINO programming connector pins 4, 5, & 6 are used as the inputs for the three logic channels. There are holes in the board where you can solder a header for connecting to the Logic Analyzer or you can route signals to the B7 and B6 pins to display the signal.



Pin	Function
1	MCLR
2	VDD
3	GND
4	(B7) Channel 1
5	(B6) Channel 2
6	AUX Channel 3

For example, to monitor a SPI bus, the analyzer channel pins could be connected to monitor the three main bus signals as follows:

Logic Analyzer Pin	SPI Bus Signal
Channel 1	SCK (bus master clock)
Channel 2	SDO (bus master output)
Channel 3	SDI (bus master input)

Note: Pin 4 and Pin 5 have a 4.7k Ohm pulldown resistor internal. This resistor is necessary for the debugger functions, but note that this pulldown resistor will affect any digital signal these pins are connected to.

The Logic Analyzer Window

The Logic Tool analyzer window is divided into 3 sections. These include:

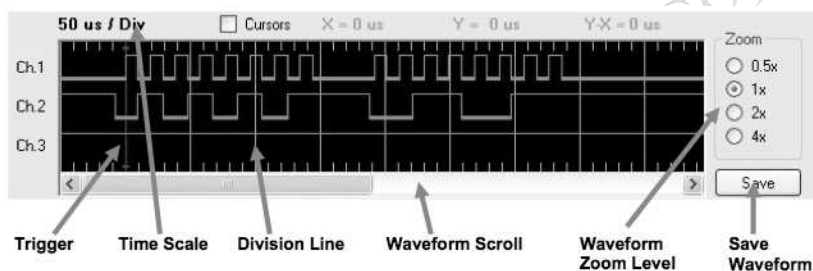
- 1) Display – for viewing and measuring captured waveforms.
- 2) Trigger – for setting trigger conditions for a capture

3) Acquisition – for setting the waveform sample rate and the waveform relation to the trigger sample.

Analyzer Display

The display section of the analyzer window allows the waveform to be viewed, zoomed, measured, and saved as a bitmap file.

Below is a SPI bus waveform capture of a 2-byte transmission, and details the elements of the display window section.



Trigger

The trigger is a pre-defined event in the monitored signals that causes a capture of the signal waveform. Triggering is discussed in detail in section 3.2.2 The Analyzer Trigger Section.

On the waveform display, the point where the trigger occurred is indicated by a vertical red line. The trigger was set to occur at the first rising edge of Channel 1, in the SPI SCLK clock signal above.

Time Scale

Above left of the waveform display is the time scale. This is how much time each Division Line in the waveform represents. Each division is 50 microseconds of time in the picture.

Division Line

A division line is a gray vertical line across the waveform display, which can be used to give a time reference to the displayed waveform. Smaller hash marks at the top and bottom of the display subdivide each time division into 5 smaller units.

Waveform Scroll

The captured waveform is longer than can be shown all at once effectively in the display, so the horizontal scroll bar allows the display to scroll for viewing the entire waveform.

Waveform Zoom Level

The waveform Zoom allows 4 levels of zoom to be selected. Normally, at 1x zoom each sample of a waveform is displayed as a pixel. A waveform is 1024 pixels, of which 500 can be displayed in the window. By selecting zoom level "0.5x", the waveform is compressed so 2 samples are shown per pixel, which allows the entire waveform to be view at once, but with a loss of detail.

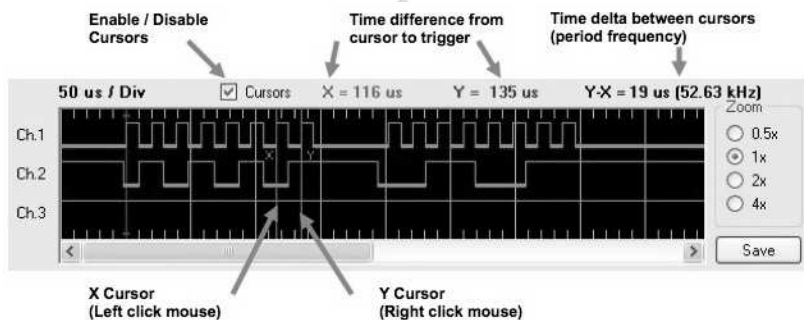
Zoom levels 2x and 4x display the waveform with 2 pixels per sample and 4 pixels per sample, respectively. This allows relative time details between the waveforms to be more easily seen.

Save Waveform

Click the Save button to save the current waveform display in a bitmap file. The time scale will be added to the bottom of the display. If cursors are active, the cursors and their times will also be saved with the display.

Analyzer Display Cursors

The display cursors are useful for making time and frequency measurements in the displayed waveform. Click to check the "Cursors" checkbox and enable the cursors.



Place the X cursor by left-clicking in the waveform display.

Place the Y cursor by right-clicking in the waveform display.

It can be helpful to use Zoom for exact placement of the cursors. When zoomed, the cursors will get "wider" as they are the width of a sample and the sample width grows with increasing zoom.

Above the waveform display, the time difference between the Trigger and each cursor is displayed, along with the difference between the triggers. The time period between the cursor is also displayed as the related frequency.

Analyzer Trigger

The “trigger” is a user-defined set of events in the monitored signals that causes the capture of a waveform. Each channel can be assigned one of the following trigger events:

Trigger

Trigger when

Ch 1 = / * - Don't Care
and 1 - Logic High
Ch 2 = 1 0 - Logic Low
and / - Rising Edge
Ch 3 = * \ - Falling Edge

occurs 4 times.
(1 - 256)

'*' (Don't Care) - The analyzer channel is ignored for triggering purposes

'1' (Logic High) - The channel must be at a logic high state to trigger

'0' (Logic Low) - The channel must be at a logic low state to trigger

'/' (Rising Edge) - The channel must transition from low to high states to trigger

'\' (Falling Edge) - The channel must transition from high to low states to trigger

It's best to set just one trigger at a time but if you set trigger events on all channels, they must happen all at once in order for the trigger to activate data capture.

For example:

These settings will trigger on the rising edge of Channel 1.

Ch 1 = / (rising edge)

Ch 2 = * (ignore)

Ch 3 = * (ignore)

These settings will trigger on the rising edge of Channel 1 and Channel 2 at a high level.

Ch 1 = / (rising edge)

Ch 2 = 1 (logic high)

Ch 3 = * (ignore)

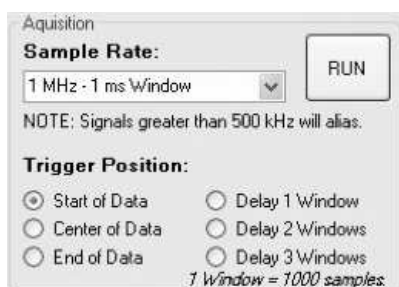
It is also possible to specify how many times the trigger condition must occur before waveform capture is initiated (up to 256 times maximum).

For example, suppose we wanted to capture the 16th byte of a long SPI transmission sequence. If we triggered on the first clock edge of the first byte, we probably wouldn't be able to see the 16th byte, as the analyzer would stop

sampling before it occurred. However, we can set the analyzer to pass up the first 15 bytes, by setting the trigger count to 15 bytes * 8 clocks + 1 = 121 times. This way, it will start counting clock edges on the first byte, but it won't trigger the data capture until the 16th byte is transmitted.

Analyzer Acquisition

The "Acquisition" section of the analyzer window is used to set the waveform sample rate, the position of the trigger relative to the captured waveform, and to start or "run" the analyzer.



Sample Rate

The sample rate is how often the analyzer channels are looked at. Each waveform capture is only 1024 samples long, so if we want to look at a longer period of time in the waveform display, we have to sample less often. The trade-off is that at higher sample rates, we can see more detail and faster signals but only a small window of time. At lower sample rates, we can see a longer window of elapsed time but at less detail and may miss fast pulses. Generally, the sample rate should be set at least 10 times the highest frequency or 5 times the fastest pulse width to get a decent representation of the waveform.

Any waveform that has frequency higher than half the sample rate may alias. Aliasing means that waveform edges are missed and so the waveform can appear slower than it actually is. Of course, the sample rate can always be set slower than these limits if all that's desired is to get a general idea of what's going on in the circuit without much detail.

SUPPORTED SAMPLE RATES

Sample Rate	Time Between Samples	Waveform Limitations	
		Captured Waveform Length ¹ (1024 samples)	Maximum Frequency (before aliasing)
1 MHz	1 us	1 ms	500 kHz
500 kHz	2 us	2 ms	250 kHz
250 kHz	4 us	4.1 ms	125 kHz
100 kHz	10 us	10.2 ms	50 kHz
50 kHz	20 us	20.5 ms	25 kHz
25 kHz	40 us	41 ms	12.5 kHz
10 kHz	100 us	102.4 ms	5 kHz
5 kHz	200 us	204.8 ms	2.5 kHz

Note 1: Waveform length is rounded to the nearest 0.1 decimal place.

Trigger Position

Changing the trigger position allows more flexibility over how the captured data relates to the trigger event. For example, we might be more interested in what happened before the trigger, rather than after.

There are 6 selectable trigger positions:

Start of Data

This is the trigger position used in all the prior Figure waveforms. All the waveform data, except for one division, is captured after the trigger occurs. This is best used when all the waveform data of interest happens after the trigger.

Center of Data

This is best used when all the waveform data of interest happens around the trigger. The trigger event is in the middle of the waveform display.

End of Data

All the waveform data, except for just over one division, is captured prior to the trigger occurs. This is best used when all the waveform data of interest happens before the trigger.

Delay 1 Window

Delay 2 Windows

Delay 3 Windows

In these cases, the trigger position is considered “Start of Data” but the waveform capture is delayed 1000 samples (nearly one waveform display) after the trigger. This allows a user to capture events that occur further out than the display width after a trigger happens, without reducing the sample rate.

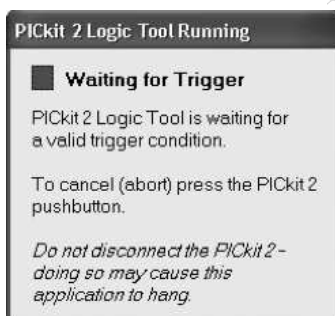
In other words, when “Delay 1 Window” is selected, the analyzer will wait 1000 samples after the trigger event occurs before it begins recording waveform data. When “Delay 2 Windows” is selected, it will wait 2000 samples etc.

Each waveform display is 1024 samples, so the 1000 sample delay increment gives a small overlap between successive delay captures. Assuming that the data of interest after the trigger is repeatable and consistent, this allows a total waveform of up to 4 times the sample rate window width to be pieced together.

For example, it would be possible to collect 4 ms worth of waveform data after a trigger event at the 1 MHz sample rate.

Running the Analyzer

Once the trigger conditions, sample rate, and trigger position are set as desired click the RUN button to begin collecting waveform data and looking for trigger events.



When the analyzer is running, it will show the dialog above and the "Busy" LED on the PICKit 2 unit will be lit.

Once the trigger condition is met the "Busy" LED will turn off, the "Waiting for Trigger" dialog will close, and the analyzer waveform display will be updated with the newly captured data.